

On the Expressiveness and Complexity of MongoDB

Elena Botoeva
Free University of
Bozen-Bolzano
botoeva@inf.unibz.it

Benjamin Cogrel
Free University of
Bozen-Bolzano
cogrel@inf.unibz.it

Diego Calvanese
Free University of
Bozen-Bolzano
calvanese@inf.unibz.it

Guohui Xiao
Free University of
Bozen-Bolzano
xiao@inf.unibz.it

ABSTRACT

A significant number of novel database architectures and data models have been proposed during the last decade. While some of these new systems have gained in popularity, they lack a proper formalization, and a precise understanding of the expressiveness and the computational properties of the associated query languages. In this paper, we aim at filling this gap, and we do so by considering MongoDB, a widely adopted document database managing complex (tree structured) values represented in a JSON-based data model, equipped with a powerful query mechanism. We provide a formalization of the MongoDB data model, and of a core fragment, called MQuery, of the MongoDB query language. We study the expressiveness of MQuery, showing its equivalence with nested relational algebra. We further investigate the computational complexity of significant fragments of it, obtaining several (tight) bounds in combined complexity, which range from LOGSPACE to alternating exponential-time with a polynomial number of alternations.

1. INTRODUCTION

As was envisioned by Stonebraker and Cetintemel [21], during the last ten years a diversity of new database (DB) architectures and data models has emerged, driven by the goal of better addressing the widely varying demands of modern data-intensive applications. Notably, many of these new systems do not rely on the relational model but instead adopt a less rigid data format, and alternative query mechanisms, which combine an increased flexibility in dealing with semi-structured data, with a higher efficiency (at least for some types of common operations). Hence the emergence of the term *NoSQL* (for “not only SQL”) [8, 17].

A large portion of the so-called *non-relational* systems (e.g., MongoDB, CouchDB, and Hadoop) organize data in collections of semi-structured, tree-shaped documents in the JavaScript Object Notation (JSON) format, which is commonly viewed as a lightweight alternative to XML. Such documents can be seen as complex values [14, 1, 26, 12], in particular due to the presence of nested arrays. As an example, consider the document in Figure 1, containing standard personal information about Kristen Nygaard (such as name and birth-date) together with information about the awards he received, the latter being stored inside an array.

Among the non-relational languages that have been proposed for querying JSON collections (see, e.g., [3, 19,

```
{ "_id": 4,
  "awards": [
    { "award": "Rosing Prize", "year": 1999,
      "by": "Norwegian Data Association" },
    { "award": "Turing Award", "year": 2001,
      "by": "ACM" },
    { "award": "IEEE John von Neumann Medal",
      "year": 2001, "by": "IEEE" } ],
  "birth": "1926-08-27",
  "contribs": ["OOP", "Simula"],
  "death": "2002-08-10",
  "name": { "first": "Kristen",
            "last": "Nygaard" }
}
```

Figure 1: A sample MongoDB document in the *bios* collection

24] and the *MongoDB aggregation framework*¹), languages with rich capabilities have unsurprisingly many similarities with well-known query languages for complex values, such as monad algebra (MA) [5, 16], nested relational algebra (NRA) [23, 25] and Core XQuery [16]. For instance, Jaql [3], one of the most prominent query language targeting map-reduce frameworks [13], supports higher-order functions, which have their roots in MA, and the group and unwind operators of the MongoDB aggregation framework are similar to the nest and unnest operators of NRA. While some of these query languages have been widely used in large-scale applications, there have been only few attempts at capturing their formal semantics, e.g., through a calculus for Jaql [2], and even less at understanding their computational properties.

In this paper, we consider the case of MongoDB, a widely adopted distributed JSON-based document database, and conduct the first major investigation into the formal foundations and properties of its data model and query language. MongoDB provides rich querying capabilities by means of the *aggregation framework*, which is modeled on the notion of data processing pipelines. In this framework, a query is a multi-stage pipeline, where each stage defines a transformation, using a MongoDB-specific operator, applied to the set of documents produced by the previous stage.

Our first contribution is a formalization of the Mon-

¹<https://docs.mongodb.com/core/aggregation-pipeline/>

goDB data model and of the fragment of the aggregation framework query language that includes the *match*, *unwind*, *project*, *group*, and *lookup* operators, and which we call *MQuery*. Each of these operators roughly corresponds to an operator of NRA: *match* corresponds to select, *project* to project, *lookup* to left join, and as mentioned above, *group* to nest, and *unwind* to unnest. As a useful side-effect of our formalization effort, we point out different “features” exhibited by MongoDB’s query language that are somewhat counter-intuitive, and that might need to be reconsidered by the MongoDB developers for future versions of the system. In our investigation, we consider various fragments of *MQuery*, which we denote by \mathcal{M}^α , where α consists of the initials of the stages that can be used in the fragment.

Our second contribution is a characterization of the expressive power of *MQuery* obtained by comparing it with NRA. We define the relational view of JSON documents, and devise translations in both directions between *MQuery* and NRA, showing that the two languages are equivalent in expressive power. We also consider the $\mathcal{M}^{\text{MUPG}}$ fragment, where we rule out the recently (in Version 3.2) added *lookup* operator, which allows for joining a given document collection with external ones. Actually, we establish that already $\mathcal{M}^{\text{MUPG}}$ is equivalent to NRA over a single relation, and hence is capable of expressing arbitrary joins (within one collection), contrary to what believed in the community of MongoDB practitioners and users. Interestingly, all our translations are compact (i.e., polynomial), hence they allow us also to carry over complexity results between *MQuery* and NRA.

Finally, we carry out an investigation of the computational complexity of $\mathcal{M}^{\text{MUPGL}}$ and its fragments. In particular, we establish that what we consider the minimal fragment, which allows only for *match*, is LOGSPACE-complete (in combined complexity). Projection and grouping allow one to create exponentially large objects, but by representing intermediate results compactly as DAGs, one can still evaluate $\mathcal{M}^{\text{MUGL}}$ queries in PTIME. The use of *unwind* alone causes loss of tractability in combined complexity, specifically it leads to NP-completeness, but remains LOGSPACE-complete in query complexity. Adding also *project* or *lookup* leads again to intractability even in query complexity, although $\mathcal{M}^{\text{MUGL}}$ stays NP-complete in combined complexity. In the presence of *unwind*, grouping provides another source of complexity, since it allows one to create doubly-exponentially large objects; indeed we show PSPACE-hardness of \mathcal{M}^{MUG} . Finally, we establish that the full language and also the $\mathcal{M}^{\text{MUPG}}$ fragment are complete for exponential time with a polynomial number of alternations (in combined complexity). As mentioned, our polynomial translations between *MQuery* and NRA, allow us to carry over the complexity results also to NRA (and its fragments). In particular, we establish a tight $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ result for the combined complexity of Boolean query evaluation in NRA, whose exact complexity was open.

The rest of the paper is structured as follows. In Section 2 we introduce NRA. In Section 3 we provide our formalization of MongoDB documents, and in Section 4 of the MongoDB query language. In Section 5 we study the expressiveness of such language by providing translations to and from

NRA, and in Section 6 we study its computational complexity. We conclude the paper in Section 7. Selected proofs are given in the Appendix.

2. PRELIMINARIES

We recap the basics of nested relational algebra (NRA) [10, 25], mainly to fix the notation of the operators. For their semantics we refer to [10].

Let \mathcal{A} be a countably infinite set of attribute names and relation schema names. A *relation schema* of a relation can have the form $R(S)$, where $R \in \mathcal{A}$ is a relation schema name and S is a finite set of attributes, each of which is an atomic attribute (i.e., an attribute name in \mathcal{A}) or a relation schema of a sub-relation. A relation schema can also be obtained through a NRA operation (see below). We use the function *att* to retrieve the attributes from a relation schema name, i.e., $\text{att}(R) = S$. A relation schema $R(S)$ is called *flat* if S consists of atomic attribute only. In the following, when convenient, we refer to relation schemas by their name only.

Let Δ be the domain of all the atomic attributes in \mathcal{A} . An *instance* of a relation schema $R(S)$ is a finite set of tuples over $R(S)$. A *tuple* t over $R(S)$ is a finite set $\{a_1:v_1, \dots, a_n:v_n\}$ such that if a_i is an atomic attribute, then $v_i \in \Delta$, and if a_i is a relation schema, then v_i is an instance of a_i .

A *filter* ψ over a set $A \subseteq \mathcal{A}$ is a Boolean formula constructed from atoms of the form $(v \in a)$, $(v = a)$ or $(a = a')$, where $\{a, a'\} \subseteq A$, and v is an atomic value or a relation. Let R and R' be relation schemas. We make of use the following relational algebra operators: (1) *set union* $R \cup R'$ and *set difference* $R \setminus R'$, for $\text{att}(R) = \text{att}(R')$; (2) *cross-product* $R \times R'$, resulting in a relation schema with attributes $\{\text{rel1}.a \mid a \in \text{att}(R)\} \cup \{\text{rel2}.a \mid a \in \text{att}(R')\}$; (3) *selection* $\sigma_\psi(R)$, where ψ is a filter over $\text{att}(R)$; (4) *projection* $\pi_A(R)$, for $A \subseteq \text{att}(R)$; (5) *extended projection* $\pi_P(S)$, where P may also contain elements of the form $b/f(a_1, \dots, a_n)$, for a computable function f and $\{a_1, \dots, a_n\} \in \text{att}(R)$.

We also use the two operators of NRA: (6) *nest* $\nu_{\{a_1, \dots, a_n\} \rightarrow b}(R)$, resulting in a schema with attributes $(\text{att}(R) \setminus \{a_1, \dots, a_n\}) \cup \{b(a_1, \dots, a_n)\}$ and (7) *unnest* $\chi_a(R)$, resulting in a schema with attributes $(\text{att}(R) \setminus \{a\}) \cup \text{att}(a)$. We note that $\chi_a(R)$ will not preserve tuples t if $\pi_a(\{t\}) = \{\}$. We also assume that in NRA the project operator π can access sub-relations at all levels as in [10].

3. MongoDB DOCUMENTS

In this section we propose a formalization of the syntax and the semantics of MongoDB documents. In our formalization, we make two simplifying assumptions with respect to the way such documents are treated by the MongoDB system: (i) we view documents as expressed in JSON, as opposed to BSON², since we abstract away document order, and (ii) we consider set-semantics as opposed to bag-semantics. In particular, we avoid deep comparison of objects, which does not follow the standard semantics but is based on their binary representation and is deprecated³.

²<https://docs.mongodb.org/manual/reference/bson-types/>

³Personal communication by the MongoDB development team.

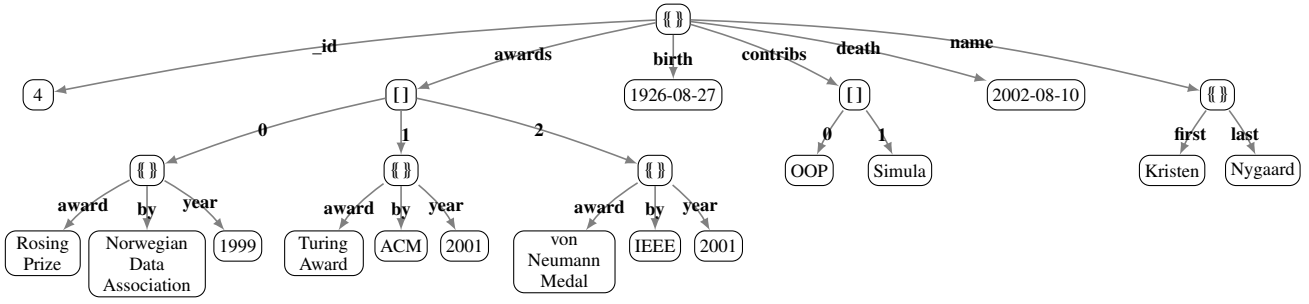


Figure 2: The tree representation of the MongoDB document in Figure 1

$\text{VALUE} ::= \text{LITERAL} \mid \text{OBJECT} \mid \text{ARRAY}$
 $\text{OBJECT} ::= \{ \text{LIST<KEY : VALUE>} \}$
 $\text{ARRAY} ::= [\text{LIST<VALUE>}]$

 $\text{LIST<T>} ::= \varepsilon \mid \text{LIST}^+ \text{<T>}$
 $\text{LIST}^+ \text{<T>} ::= \text{T} \mid \text{T}, \text{LIST}^+ \text{<T>}$

Figure 3: Syntax of JSON objects

A MongoDB database stores collections of documents, where each document is an object consisting of key-value pairs⁴, and a value can itself be a nested object. A collection corresponds to a table in a (nested) relational database, and a document corresponds to a row in a table.

We start by defining the syntax of MongoDB documents in the JSON format. *Literals* are atomic values, such as strings, numbers, and booleans. A *JSON object* is a finite set of key-value pairs, where a *key* is a string and a *value* can be a literal, an object, or an array of values, constructed according to the grammar in Figure 3 (where terminals are written in black, and non-terminals in blue). We require that the set of key-value pairs constituting a JSON object does not contain the same key twice. A (*MongoDB*) *document* is a JSON object (not nested within any other object) with a special key ‘_id’, which is used to identify the document. Figure 1 shows a MongoDB document in which, apart from _id, the keys are birth, name, awards, etc. Notice that the value of name is an object consisting of two key-value pairs, and the value of awards is an array of objects, each describing an award. Given a collection name C , a (*MongoDB*) *collection* for C is a finite set F_C of documents, such that each document is identified by its value of _id, i.e., each value of _id is unique in F_C . Given a set \mathbb{C} of collection names, a *MongoDB database instance* D (over \mathbb{C}) is a set of collections, one for each name $C \in \mathbb{C}$. We write $D.C$ to denote the collection for name C .

We formalize MongoDB documents as finite *unordered, unranked, node-labeled, and edge-labeled trees*. We assume three disjoint sets of labels: the sets K of *keys* and I of *indexes* (non-negative integers), used as edge-labels, and the set V of *literals*, containing the special elements **null**, **true**, and **false**, and used as node labels. A *tree* is a tuple (N, E, L_n, L_e) , where N is a set of nodes, E is a successor

relation, $L_n : N \rightarrow V \cup \{ \{ \} , [] \}$ is a node labeling function, and $L_e : E \rightarrow K \cup I$ is an edge labeling function, such that (i) (N, E) forms a tree, (ii) a node labeled by a literal must be a leaf, (iii) all outgoing edges of a node labeled by ‘{ }’ must be labeled by keys, and (iv) all outgoing edges of a node labeled by ‘[]’ must be labeled by distinct indexes. Given a tree t and a node x , the *type* of x in t , denoted $\text{type}(x, t)$, is literal if $L_n(x) \in V$, object if $L_n(x) = \{ \}$, and array if $L_n(x) = []$. The root of t is denoted by $\text{root}(t)$. A *forest* is a set of trees. If $\text{root}(t)$ has an outgoing edge labeled with _id, we call the tree t a *document*.

Given a tree t , we define inductively for each node x in t , the *value represented by x in t* , denoted $\text{value}(x, t)$, as follows: (i) if x is a leaf in t , then $\text{value}(x, t) = L_n(x)$; (ii) let x_1, \dots, x_m be all the children of x with the corresponding edges labeled by k_1, \dots, k_m . If $\text{type}(x, t) = \text{object}$, then $\text{value}(x, t) = \{ k_1 : \text{value}(x_1, t), \dots, k_m : \text{value}(x_m, t) \}$, and if $\text{type}(x, t) = \text{array}$, then $\text{value}(x, t) = [\text{value}(x_1, t), \dots, \text{value}(x_m, t)]$. The *JSON document represented by t* , denoted $\text{value}(t)$, is then $\text{value}(\text{root}(t), t)$.

The *tree corresponding to a value u* , denoted $\text{tree}(u)$, is defined as (N, E, L_n, L_e) , where N is the set of x_v such that v is an object, array, or literal value appearing in u , and for $x_v \in N$: (i) if v is a literal, then $L_n(x_v) = v$ and x_v is a leaf; (ii) if $v = \{ k_1 : v_1, \dots, k_m : v_m \}$, for $m \geq 0$, then $L_n(x_v) = \{ \}$, x_v has m children x_{v_1}, \dots, x_{v_m} with $L_e(x_v, x_{v_i}) = k_i$; (iii) if $v = [v_1, \dots, v_m]$, for $m \geq 0$, then $L_n(x_v) = []$, x_v has m children x_{v_1}, \dots, x_{v_m} with $L_e(x_v, x_{v_i}) = i - 1$. Observe that a literal v can be seen as a tree consisting of a single node whose label is v . Then, the tree corresponding to a JSON document d is defined as $\text{tree}(d)$, where d is viewed as a value. The tree representation of the document in Figure 1 is depicted in Figure 2.

4. MongoDB QUERIES

The basic query mechanism of MongoDB are so-called *find* queries, which are subsumed by the more powerful query mechanism provided by the *aggregation framework*. In this paper, we present and formalize only the query language derived from the aggregation framework, and we refer to this query language (or rather, family of languages) as *MQuery*.

4.1 Syntax of MQuery

An *MQuery* is a sequence of stages, also called a *pipeline*,

⁴Here we adopt the same terminology as MongoDB, where the term ‘key’ is used with the meaning of ‘attribute’ in relational databases, not to be confused with the traditional notion in ‘key constraints’.

$\begin{aligned} \text{MAQ} &::= \text{COLLECTION}.\text{aggregate}([\text{LIST}^+ \langle \text{STAGE} \rangle]) \\ \text{STAGE} &::= \{ \$\text{match}: \{ \text{CRITERION} \} \} \mid \{ \$\text{unwind}: \{ \text{UNWINDEXPR} \} \} \mid \{ \$\text{project}: \{ \text{PROJECTION} \} \} \\ &\quad \mid \{ \$\text{group}: \{ \text{GROUPEXPR} \} \} \mid \{ \$\text{lookup}: \{ \text{LOOKUPEXPR} \} \} \end{aligned}$	
$\begin{aligned} \text{PATH} &::= \text{KEY} \mid \text{KEY}.\text{PATH} \\ \text{PATHREF} &::= \$\text{PATH} \mid \$\text{\$ROOT} \\ \text{LOP} &::= \$\text{and} \mid \$\text{or} \mid \$\text{nor} \\ \text{COP} &::= \$\text{eq} \mid \$\text{gt} \mid \$\text{lt} \\ &\quad \mid \$\text{ne} \mid \$\text{gte} \mid \$\text{lte} \\ \text{BOP} &::= \text{LOP} \mid \text{COP} \\ \text{BOOLEAN} &::= \text{true} \mid \text{false} \\ \text{CRITERION} &::= \text{PATH} : \text{CONDITION} \\ &\quad \mid \text{LOP} : [\text{LIST}^+ \langle \{ \text{CRITERION} \} \rangle] \\ \text{CONDITION} &::= \{ \text{COP} : \text{VALUE} \} \\ &\quad \mid \{ \$\text{not} : \text{CONDITION} \} \\ &\quad \mid \{ \$\text{exists} : \text{BOOLEAN} \} \\ \text{PROJECTION} &::= \text{LIST}^+ \langle \text{PROJECTIONELEM} \rangle \\ \text{PROJECTIONELEM} &::= _id : \text{false} \\ &\quad \mid \text{PATH} : \text{true} \\ &\quad \mid \text{PATH} : \text{VALUEDEF} \end{aligned}$	$\begin{aligned} \text{VALUEDEF} &::= \text{PATHREF} \\ &\quad \mid \{ \$\text{literal} : \text{VALUE} \} \\ &\quad \mid [\text{LIST} \langle \text{VALUEDEF} \rangle] \\ &\quad \mid \{ \text{BOP} : [\text{LIST} \langle \text{VALUEDEF} \rangle] \} \\ &\quad \mid \{ \$\text{not} : \text{VALUEDEF} \} \\ &\quad \mid \{ \$\text{cond} : \{ \text{if} : \text{VALUEDEF}, \\ &\quad \quad \text{then} : \text{VALUEDEF}, \\ &\quad \quad \text{else} : \text{VALUEDEF} \} \} \\ \text{GROUPEXPR} &::= _id : \text{GROUPCONDITION}, \\ &\quad \text{LIST} \langle \text{KEY} : \{ \$\text{addToSet} : \text{PATHREF} \} \rangle \\ \text{GROUPCONDITION} &::= \text{null} \mid \{ \text{LIST} \langle \text{PATH} : \text{PATHREF} \rangle \} \\ \text{UNWINDEXPR} &::= \text{path} : \text{PATHREF}, \\ &\quad \text{preserveNullAndEmptyArrays} : \text{BOOLEAN} \\ \text{LOOKUPEXPR} &::= \text{from} : \text{COLLECTION}, \\ &\quad \text{localField} : \text{PATH}, \\ &\quad \text{foreignField} : \text{PATH}, \\ &\quad \text{as} : \text{PATH} \end{aligned}$

Figure 4: The MQuery grammar

applied to a collection name, where each of the stages transforms a forest into another forest.

The grammar for the MQuery language is presented in Figure 4. MQuery allows for five types of stages: (i) *match* μ , which selects trees of interest, (ii) *unwind* ω , which flattens an array from the input trees to output a tree for each element of the array, (iii) *project* ρ , which modifies trees by projecting away paths, renaming paths, or introducing new paths, (iv) *group* γ , which groups trees according to the values of a set of paths, and (v) *lookup* λ , which joins trees in the pipeline with trees in an external collection C , using a local path and a path in C to express the join condition, and an additional path to store the matching trees. We consider also various fragments of MQuery, and we denote each fragment by \mathcal{M}^α , where α consists of the initials of the stages that can be used in queries in the fragment. Hence, $\mathcal{M}^{\text{MUPGL}}$ denotes MQuery itself, and, e.g., $\mathcal{M}^{\text{MUPG}}$ denotes the fragment of $\mathcal{M}^{\text{MUPGL}}$ that does not use lookup, and \mathcal{M}^{MUP} the fragment that additionally does not use group. We observe that MongoDB find queries correspond to a simple case of \mathcal{M}^{MP} , where a match stage is followed by a project stage in which each projection element is of the form $\text{PATH} : \text{true}$ and $_id : \text{false}$.

We provide some comments and additional requirements on the grammar in Figure 4. A **PATH** (which in MongoDB terminology is actually called a “field”), is a non-empty concatenation of **KEYS**, where elements for **KEY** are from the set K . Elements for **VALUE** are defined according to the grammar in Figure 3. **COLLECTION** is a collection name, that is, a non-empty string. The empty path, which can be used in a path reference, is denoted in MongoDB by the string $\text{\$ROOT}$. In the following, a *path* is either the empty path or an element constructed according to **PATH**. For two paths p and p' , we say that p' is a *strict prefix* of p , if $p = p'.p''$, for some non-empty path p'' . Also, p' is a *prefix* of p if p' is either a strict prefix of p or equal to p . We assume that a projection $p_1:d_1, \dots, p_n:d_n$ is such that there are no $i \neq j$ where p_i is a prefix of p_j . By default the $_id$ key is kept

$$\begin{aligned} \varphi &::= p = v \mid \exists p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg \varphi \\ d &::= p \mid v \mid [d, \dots, d] \mid \beta \mid \kappa \\ \beta &::= p = p \mid p = v \mid v = v \mid \exists p \mid d \vee d \mid d \wedge d \mid \neg d \\ \kappa &::= (d?d:d) \\ \hline s &::= \mu_\varphi \mid \omega_p^n \mid \rho_P^n \mid \gamma_{G:A} \mid \lambda_p^{p_1=C.p_2} \\ \text{MQuery} &::= C \triangleright s \triangleright \dots \triangleright s \end{aligned}$$

Figure 5: Algebra for MQuery

in a projection, and to project it away the projection must contain an element $_id : \text{false}$. The comparison operators used in a value definition **VALUEDEF** accept only arrays of length 2. We observe that, with respect to the official MongoDB syntax, we have removed/introduced some syntactic sugar. In particular, for **CRITERION** we disallow expressions of the form “name.first”: “john”. Instead they can be expressed as “name.first”: { \$eq: “john” }. Moreover, we allow for the use of $\$nor$ in **VALUEDEF**, as it can be expressed using $\$not$ and $\$and$.

4.2 Semantics of MQuery

To abstract away syntactic aspects of MQuery, and allow us to formalize its semantics, we first propose for it an algebra, shown in Figure 5. In this algebra, p stands for a path, v for a value, φ for a criterion, d for a value definition, β for a Boolean value definition, κ for a conditional value definition, s for a stage, and C for a collection name. For simplicity of presentation, the only comparison operator that we kept in the algebra is equality. Adding also order comparison would not affect any of the results on expressiveness and complexity presented later. Moreover, we denote the query stages as follows:

- μ_φ for a match stage;
- ω_p for an unwind stage that does not preserve null and empty arrays, and ω_p^+ for an unwind stage that preserves

Match	$t \models (p = v), \quad \text{if there is } x \text{ in } \llbracket p \rrbracket^t \text{ or } \llbracket p.i \rrbracket^t \text{ for } i \in I, \text{ with } \models (\text{value}(x, t) = v).$ $t \models (\exists p), \quad \text{if } \llbracket p \rrbracket^t \neq \emptyset.$ $t \models \varphi_1 \wedge \varphi_2, \quad \text{if } t \models \varphi_1 \text{ and } t \models \varphi_2.$ $t \models \varphi_1 \vee \varphi_2, \quad \text{if } t \models \varphi_1 \text{ or } t \models \varphi_2.$ $t \models \neg \varphi, \quad \text{if } t \not\models \varphi.$ $F \triangleright \mu_\varphi = \{t \mid t \in F \text{ and } t \models \varphi\}$
Project	$t \models (p_1 = p_2) \text{ if there is a value } v \text{ such that } t \models (p_1 = v) \text{ and } t \models (p_2 = v), \text{ or } t \models \neg(\exists p_j), j = 1, 2.$ $t \models (v_1 = v_2) \text{ if } \models (v_1 = v_2). \quad t \models v, \text{ for a value } v, \text{ if } v \notin \{\text{null}, \text{false}, 0\}.$ $t \models [d_1, \dots, d_n] \text{ always.} \quad t \models p, \text{ for a path } p, \text{ if } t \models (\exists p) \wedge \neg(p = v) \text{ for } v \in \{\text{null}, \text{false}, 0\}.$ $t \models (c?d_1:d_2) \text{ if } t \models c \text{ and } t \models d_1, \text{ or if } t \not\models c \text{ and } t \models d_2.$ $\rho_p(t) = \text{subtree}(t, N_p), \text{ where } N_p \text{ are the nodes in } t \text{ on a path from root}(t) \text{ to a leaf through some } x \in \llbracket p \rrbracket^t$ $\rho_{q/p}(t) = \begin{cases} \text{attach}(q, \text{tree}(v_p)), & \text{if } t \models \exists p \\ \emptyset, & \text{otherwise} \end{cases} \quad \rho_{q/d}(t) = \text{attach}(q, \text{tree}(v_d))$ $\rho_{q/(c?d_1:d_2)}(t) = \rho_{q/d}(t) \text{ where } d \text{ is } d_1 \text{ if } t \models c, \text{ and } d \text{ is } d_2 \text{ otherwise}$ $\rho_P(t) = \text{subtree}(t, _id) \oplus \bigoplus_{p \in P} \rho_p(t) \oplus \bigoplus_{(q/d) \in P} \rho_{q/d}(t) \quad \rho_P^{\neg \exists \varphi}(t) = \bigoplus_{p \in P} \rho_p(t) \oplus \bigoplus_{(q/d) \in P} \rho_{q/d}(t)$ $F \triangleright \rho_P^{ni} = \{\rho_P^{ni}(t) \mid t \in F\}$
Unwind	$\omega_p(t) = \begin{cases} \{(t \setminus \text{subtree}(t, p)) \oplus \text{attach}(p, \text{subtree}(t, p.i))\}_{i \in I, \llbracket p.i \rrbracket^t \neq \emptyset}, & \text{if } p \text{ is a first level array,} \\ \emptyset, & \text{otherwise.} \end{cases}$ $\omega_p^+(t) = \begin{cases} \omega_p(t), & \text{if } \omega_p(t) \neq \emptyset, \\ \{t\}, & \text{otherwise.} \end{cases}$ $F \triangleright \omega_p^n = \bigcup_{t \in F} \omega_p^n(t)$
Group	$F \triangleright \gamma_{\text{null}:a_1/b_1, \dots, a_m/b_m} = \{\text{attach}(_id, \text{null}) \oplus \bigoplus_{j=1..m} \text{attach}(a_j, \text{array}(F, b_j))\}$ $F \triangleright \gamma_{g_1/y_1, \dots, g_n/y_n: a_1/b_1, \dots, a_m/b_m} = \left\{ \text{attach}(_id, \{\}) \oplus \bigoplus_{j=1..m} \text{attach}(a_j, \text{array}(F \triangleright \mu_\varphi, b_j)) \mid \varphi = \bigwedge_{i=1..n} (\neg \exists y_i), (F \triangleright \mu_\varphi) \neq \emptyset \right\} \cup$ $\left\{ \bigoplus_{i \in I} \text{attach}(_id.g_i, t_i) \oplus \bigoplus_{j=1..m} \text{attach}(a_j, \text{array}(F \triangleright \mu_\varphi, b_j)) \mid I \in 2^{\{1, \dots, n\}} \setminus \emptyset, \right.$ $\left. t_i \in \text{forest}(F, y_i) \text{ for } i \in I, \varphi = \bigwedge_{i \in I} ((y_i = t_i) \wedge \exists y_i) \wedge \bigwedge_{i \notin I} (\neg \exists y_i), (F \triangleright \mu_\varphi) \neq \emptyset \right\}$
Lookup	$\lambda_P^{p_1=C.p_2}[F'](t) = t \oplus \text{attach}(p, \text{array}(F' \triangleright \mu_{p_2=v_1}, \epsilon)), \text{ where } v_1 = \text{value}(\text{subtree}(t, p_1)).$ $F \triangleright \lambda_P^{p_1=C.p_2}[F'] = \{\lambda_P^{p_1=C.p_2}[F'](t) \mid t \in F\}$

Figure 6: The semantics of MQuery stages

- them;
- ρ_P for a project stage, where P is a sequence of elements of the form p or q/d , where p is a path to be kept, and q is a new path with value definition d . We observe that we included $\exists p$ as an atomic Boolean value definition β since it can be expressed using a conditional value definition (see Appendix B.1). When the $_id$ must be omitted, we write $\rho_P^{\neg \exists \varphi}$;
 - $\gamma_{G:A}$ for a group stage, where the group condition G and the aggregation paths A are (possibly empty) sequences of elements of the form p/p' , for paths p and p' , and p must be a key in A . In these sequences, if p coincides with p' , then we simply write p instead of p/p . When G is the empty sequence, it corresponds to the grouping condition being **null**; and
 - $\lambda_P^{p_1=C.p_2}$ for a lookup stage, where p_1 is the local path, p_2 is the path from the external collection C , and p is the path to store the matching trees.

Finally, we use ε to denote the empty path.

Our semantics of the MQuery algebra is based on sets. Before introducing it formally, we show how to interpret paths

over trees.

DEFINITION 4.1. *Given a tree $t = (N, E, L_n, L_e)$, we interpret a (possibly empty) path p , and its concatenation $p.i_1 \dots i_m$ with indexes i_1, \dots, i_m , as sets of nodes as follows, where k is a key:*

$$\begin{aligned} \llbracket \varepsilon \rrbracket^t &= \{\text{root}(t)\} \\ \llbracket p.k \rrbracket^t &= \{y \in N \mid \text{there exist } i_1, \dots, i_m, m \geq 0, \\ &\quad \text{and } x \in \llbracket p.i_1 \dots i_m \rrbracket^t \text{ s.t.} \\ &\quad (x, y) \in E \text{ and } L_e(x, y) = k\} \end{aligned}$$

$$\llbracket p.i_1 \dots i_m \rrbracket^t = \{y \in N \mid \text{there is } x \in \llbracket p.i_1 \dots i_{m-1} \rrbracket^t \text{ s.t.} \\ (x, y) \in E \text{ and } L_e(x, y) = i_m\}$$

When $\llbracket p \rrbracket^t = \emptyset$, we say that the path p is missing in t .

Observe that, in the above definition, the semantics of paths allows for skipping over intermediate arrays at every step in the path.

Given a tree t and a path p , when $\text{type}(x, t) = \text{ty}$, for each $x \in \llbracket p \rrbracket^t$, where $\text{ty} \in \{\text{array}, \text{literal}, \text{object}\}$, we define the *type of p in t* , denoted $\text{type}(p, t)$, to be ty . Also, when $\text{type}(p, t) = \text{array}$ and $\text{type}(x, t) = \text{ty}$ for each $x \in \llbracket p.i \rrbracket^t$

for $i \in I$, we write $\text{type}(p[i], t) = \text{ty}$. We say that p is a *first level array* in t if $\text{type}(p, t) = \text{array}$ and $\text{type}(p', t) \neq \text{array}$, for each strict prefix p' of p .

In Figure 6, we define the semantics of the MQuery stages: specifically, given a forest F and a stage s , we define the forest $F \triangleright s$, depending on the form of s (for a lookup stage, we also require an additional forest F' as parameter). For the match and project stages, we define when a tree t satisfies a criterion φ , denoted $t \models \varphi$, or a value definition d , denoted $t \models d$. In this definition we assume that for each pair of values v_1 and v_2 , the comparison $(v_1 = v_2)$ evaluates to a Boolean value. We write $t \models (v_1 = v_2)$ when $(v_1 = v_2)$ evaluates to true.⁵ In our formalization, we employ the classical semantics for “deep” comparison of non-atomic values, which differs from the actual semantics exhibited by MongoDB based on comparing the binary representation of values⁶.

To define the semantics of the *unwind*, *project*, *group*, and *lookup* operators, we make use of a number of auxiliary operators over trees, which we informally introduce here (a formal definition is given in Appendix B). Let t, t_1, t_2 be trees, F a forest, p a path, N a set of nodes, and x a node. Then: (i) $\text{subtree}(t, N)$ returns the subtree of t induced by N ; (ii) $\text{subtree}(t, p)$ returns the subtree of t hanging from a path p . In the case where $\|p\|^t > 1$, it returns the array of single subtrees, and in the case where $\|p\|^t = \emptyset$, it returns **null**; (iii) $\text{attach}(p, t)$ constructs a new tree by attaching a path p on top of the root of t ; (iv) $t_1 \setminus t_2$ returns the tree resulting from removing the subtree t_2 from t_1 ; (v) $t_1 \oplus t_2$ constructs a new tree resulting from merging t_1 and t_2 by identifying nodes reachable via identical paths; and (vi) $\text{array}(F, p)$ constructs a new tree that is the array of all $\text{subtree}(t, p)$ for $t \in F$, while $\text{forest}(F, p)$ keeps all $\text{subtree}(t, p)$ in a set.

For a value definition d (and a tree t), we denote by v_d the value associated to d in t , defined as d if $d \in V$, as $\text{value}(\text{subtree}(t, d))$ if d is a path, as the value of $(t \models d)$ if d is a Boolean value definition, and as $[v_{d_1}, \dots, v_{d_m}]$ if $d = [d_1, \dots, d_m]$.

Finally, we are ready to define the semantics of MQuery, obtained by composing (via \triangleright) the answers of its stages.

DEFINITION 4.2. Let $q = C \triangleright s_1 \triangleright \dots \triangleright s_n$ be an MQuery. The result of evaluating q over a MongoDB instance D , denoted $\text{ans}_{\text{mo}}(q, D)$, is defined as F_n , where $F_0 = D.C$, and for $i \in \{1, \dots, n\}$, $F_i = (F_{i-1} \triangleright s_i)$ if s_i is not a lookup stage, and $F_i = (F_{i-1} \triangleright s_i[D.C'])$ if s_i is a lookup stage from a collection name C' .

We illustrate the semantics of MQuery in the following examples.

EXAMPLE 4.3. Consider the tree t in Figure 2. Then t satisfies the criterion

$(\text{aws.award} = \text{"TA"}) \wedge (\text{aws.by} = \text{"IEEE"}) \wedge (\text{aws.year} = 2001)$,

but not $(\text{aws} = \{\text{award: "TA", by: "IEEE", year: 2001}\})$, where **aws** and “TA” stand for **awards** and “Turing Award”, respectively. ■

⁵We observe that $t \models (v = \text{null})$ iff v is **null**.

⁶<https://docs.mongodb.org/manual/reference/bson-types/#comparing-values>, by developing translations in both directions.

EXAMPLE 4.4. Consider the forest F :

$\{\text{tree}(\{_\text{id}: 1, \text{a}: \text{"a1"}\}), \text{tree}(\{_\text{id}: 2, \text{a}: \text{"a2"}, \text{d}: \text{"d2"}\})\}$.

Then the result of $F \triangleright \rho_{x/((_\text{id}=1)?a:c), y/((_\text{id}=2)?a:c)}$ is

$\{\text{tree}(\{_\text{id}: 1, x: \text{"a1"}\}), \text{tree}(\{_\text{id}: 2, y: \text{"a2"}\})\}$,

that is, in the first tree, the conditional value definition renames the key **a** to **x**, while in the second tree, it renames it to **y**. Note that in the else part of the conditional value definition we use a path **c** that does not appear in F : because of this and because of the semantics of *project*, **y** does not exist in the first tree, while **x** does not exist in the second tree.

Moreover, the result of evaluating $F \triangleright \gamma_{\text{null:ids}/_\text{id}}$ is

$\{\text{tree}(\{_\text{id}: \text{null}, \text{ids}: [1, 2]\})\}$

and the result of evaluating $F \triangleright \gamma_{\text{d:a}}$ is

$\{\text{tree}(\{_\text{id}: \{\}, \text{a}: [\text{"a1"}]\}), \text{tree}(\{_\text{id}: \{\text{d}: \text{"d2"}\}, \text{a}: [\text{"a2"}]\})\}$.

Note that the first tree is put into a group with $_\text{id} = \{\}$ because the path **d** is not present there. ■

Notes on our Semantics

We conclude this section by discussing some of the features in which our semantics differs from the current version of the MongoDB system. The reason for this divergence is that with respect to these features, the behavior of MongoDB might be considered counterintuitive, or even as an inconsistency in the semantics of operators.

Group. In MongoDB, the group operator behaves differently when grouping by one path and when grouping by multiple paths. In the former case **missing** is treated as **null**, while in the latter case it is treated differently. More specifically, when grouping by one path (e.g., $\gamma_{g/y:\dots}$), MongoDB puts the trees with $y = \text{null}$ and those where y is missing into the same group with $_\text{id} = \{g : \text{null}\}$. On the contrary, when grouping with multiple paths (e.g., $\gamma_{g_1/y_1, \dots, g_2/y_2:\dots}$), the trees with all y_i missing are put into a separate group with $_\text{id} = \{\}$.

Comparing value and path. The criteria in *match* and Boolean value definitions in *project* behave differently. For instance, when comparing a path p of type array with a value v using equality, match checks (1) whether v is exactly the array value of p , or (2) whether v is an element inside the array value of p . Instead, *project* only checks condition (1). More generally, for *match*, $t \models (p = v)$ if there is a node x in $\|p\|^t$ or in $\|p.i\|^t$ for some $i \in I$ such that $\text{value}(x, t) = v$, but for *project*, $t \models (p = v)$ if $\text{tree}(v)$ coincides with $\text{subtree}(t, p)$.

Null and missing values. Moreover, for *match*, $(p = \text{null})$ holds (a) when p exists and its value is **null**, or (b) when p is missing. Instead, for *project*, $(p = \text{null})$ holds only for (a).

5. EXPRESSIVENESS OF MQuery

In this section we characterize the expressiveness of MQuery in terms of nested relational algebra (NRA). More precisely, we show that MQuery is actually equivalent to NRA, by developing translations in both directions.

Q	$\text{nra2mq}(Q)$	Q	$\text{nra2mq}(Q)$
C	$\rho_{\text{att}(C)}$	$Q_1 \times Q_2$	$\text{pipeline}(\text{nra2mq}(Q_1), \text{nra2mq}(Q_2)) \triangleright$ $\gamma_{\text{rel1}, \text{rel2}} \triangleright \omega_{\text{rel1}} \triangleright \omega_{\text{rel2}}$
$\sigma_\psi(Q)$	$\text{nra2mq}(Q) \triangleright$ $\rho_{\text{att}(Q), \text{cond}/\psi} \triangleright \mu_{\text{cond}=\text{true}} \triangleright \rho_{\text{att}(Q)}$	$Q_1 \cup Q_2$	$\text{pipeline}(\text{nra2mq}(Q_1), \text{nra2mq}(Q_2)) \triangleright$ $\rho_{\text{rel1}, \text{rel2}, \{\text{pi}/((\text{actRel}=1)?\text{rel1}.\text{pi}:\text{rel2}.\text{pi})\}_{i=1}^n} \triangleright$ $\gamma_{p1, \dots, pn} \triangleright \rho_{\{\text{pi}/\text{id}.\text{pi}\}_{i=1}^n}^{\pm \text{id}}$
$\pi_S(Q)$	$\text{nra2mq}(Q) \triangleright \rho_S$	$Q_1 \setminus Q_2$	$\text{pipeline}(\text{nra2mq}(Q_1), \text{nra2mq}(Q_2)) \triangleright$ $\rho_{\text{rel1}, \text{rel2}, \{\text{pi}/((\text{actRel}=1)?\text{rel1}.\text{pi}:\text{rel2}.\text{pi})\}_{i=1}^n} \triangleright$ $\gamma_{p1, \dots, pn; \text{rel2}} \triangleright \mu_{\text{rel2}=\emptyset} \triangleright \rho_{\{\text{pi}/\text{id}.\text{pi}\}_{i=1}^n}^{\pm \text{id}}$
$\nu_{S \rightarrow b}(Q)$	$\text{nra2mq}(Q) \triangleright \rho_{(\text{att}(Q) \setminus S), \{b.p/p \mid p \in S\}} \triangleright$ $\gamma_{(\text{att}(Q) \setminus S):b} \triangleright \rho_{b, \{p/\text{id}.\text{p} \mid p \in \text{att}(Q) \setminus S\}}^{\pm \text{id}}$		
$\chi_a(Q)$	$\text{nra2mq}(Q) \triangleright \omega_a \triangleright \rho_{\text{att}(\chi_a(Q))}$		

Figure 8: Translation from NRA to $\mathcal{M}^{\text{MUPG}}$

5.2 From NRA to MQuery

We now show that NRA can be fully captured by $\mathcal{M}^{\text{MUPGL}}$, while $\mathcal{M}^{\text{MUPG}}$ captures NRA over a single collection.

In our translation from NRA to MQuery, we have to deal with the fact that an NRA query in general has a tree structure where the leaves are relation names, while an MQuery consists of a sequence of stages. So, we first show how to “linearize” tree-shaped NRA expressions into a MongoDB pipeline. More precisely, we show that it is possible to combine two $\mathcal{M}^{\text{MUPG}}$ queries q_1 and q_2 into a single $\mathcal{M}^{\text{MUPG}}$ query $\text{pipeline}(q_1, q_2)$ so that the results of q_1 and q_2 can be accessed from the result of $\text{pipeline}(q_1, q_2)$ for further processing (here we actually assume that q_1 and q_2 are sequences of stages). The idea of $\text{pipeline}(q_1, q_2)$ is to duplicate each tree t in the input forest as t_1 and t_2 so that $t_i \models (\text{actRel} = i)$, for $i \in \{1, 2\}$, and the copy of t is stored in t_i under the key $\text{rel}i$, and then to execute q_i ’s one after another:

$$\begin{aligned} \text{pipeline}(q_1, q_2) &= \rho_{\text{origDoc}/\varepsilon, \text{actRel}/[1,2]} \triangleright \omega_{\text{actRel}} \triangleright \\ &\quad \rho_{\text{actRel}, \{\text{reli}/((\text{actRel}=i)?\text{origDoc}:\text{dummy})\}_{i=1,2}} \triangleright \\ &\quad \triangleright \text{subq}_1(q_1) \triangleright \text{subq}_2(q_2) \end{aligned}$$

where dummy is a path that does not exist in any collection, and each $\text{subq}_j(q_j)$ implements the subquery q_j so as not to affect the result of the other subquery q_{3-j} . Specifically, $\text{subq}_j(q_j) = \text{subq}_j(s_1) \triangleright \dots \triangleright \text{subq}_j(s_n)$, for $q_j = s_1 \triangleright \dots \triangleright s_n$, $j \in \{1, 2\}$, and subq_j for single stages is defined as follows:

s	$\text{subq}_j(s)$
μ_φ	$\mu_{(\text{actRel} \neq j) \vee \varphi[p \rightarrow \text{rel}j.p]}$
ω_p^+	$\omega_{\text{rel}j.p}^+$
ω_p	$\mu_{(\text{actRel} \neq j) \vee ((\exists \text{rel}j.p) \wedge (\text{rel}j.p \neq []))} \triangleright \omega_{\text{rel}j.p}^+$
$\rho_{p, q/d}$	$\rho_{\text{rel}(3-j), \text{actRel}, \text{rel}j.\text{id}, \text{rel}j.p, \text{rel}j.q/((\text{actRel}=j)?d_{[\text{q}' \rightarrow \text{rel}j.q']}: \text{dummy})}$
$\gamma_{g/y:a/b}$	$\gamma_{\text{rel}j.g/\text{rel}j.y, \text{actRel}:\text{rel}j.a/\text{rel}j.b, \text{rel}(3-j)} \triangleright$ $\rho_{\text{rel}(3-j), \text{actRel}/\text{id}.\text{actRel}, \text{rel}j.a, \text{rel}j.\text{id}.\text{g}/\text{id}.\text{rel}j.g}^{\pm \text{id}} \triangleright$ $\rho_{\text{actRel}, \{\text{reli}/((\text{actRel}=i)?\text{reli}:\text{dummy})\}_{i=1,2}} \triangleright$ $\omega_{\text{rel}(3-j)}^+$

where $e_{[p \rightarrow q]}$ denotes the expression e in which every occurrence of the path p is replaced by the path q .

EXAMPLE 5.8. Consider the sequences of stages $q_1 = \mu_{\text{name.first}=\text{"John"}} \triangleright \rho_{\text{name}}$ and $q_2 = \mu_{\exists \text{awards}} \triangleright \rho_{\text{awards}}$. Then

$\text{pipeline}(q_1, q_2)$ is the following sequence of stages:

$$\begin{aligned} &\rho_{\text{origDoc}/\varepsilon, \text{actRel}/[1,2]} \triangleright \omega_{\text{actRel}} \triangleright \\ &\rho_{\text{actRel}, \{\text{reli}/((\text{actRel}=i)?\text{origDoc}:\text{dummy})\}_{i=1,2}} \triangleright \\ &\mu_{(\text{actRel} \neq 1) \vee (\text{rel1.name.first}=\text{"John"})} \triangleright \rho_{\text{rel2}, \text{actRel}, \text{rel1}.\text{id}, \text{rel1.name}} \triangleright \\ &\mu_{(\text{actRel} \neq 2) \vee (\exists \text{rel2.awards})} \triangleright \rho_{\text{rel1}, \text{actRel}, \text{rel2}.\text{id}, \text{rel2.awards}} \end{aligned}$$

Let t be the tree in Figure 2. The result of $\{t\} \triangleright \text{pipeline}(q_1, q_2)$ consists of two trees:

```
{ "actRel": 1,
  "rel1": {
    "_id": 4,
    "name": { "first": "Kristen",
              "last": "Nygaard" }
  }
}
```

and

```
{ "actRel": 2,
  "rel2": {
    "_id": 4,
    "awards": [
      { "award": "Rosing Prize", "year": 1999,
        "by": "Norwegian Data Association" },
      { "award": "Turing Award", "year": 2001,
        "by": "ACM" },
      { "award": "IEEE John von Neumann Medal",
        "year": 2001, "by": "IEEE" } ]
  }
}
```

So, the result of $\{t\} \triangleright q_1$ is found in the first tree under the key rel1 , and the result of $\{t\} \triangleright q_2$ is found in the second tree under the key rel2 . ■

We start with a singleton set $\mathcal{S} = \{(C, \tau)\}$ of type constraints for a collection name C , and consider an NRA query Q over the relation C (with schema $\text{rschema}(\tau_C)$). The translation of Q is the $\mathcal{M}^{\text{MUPG}}$ query $C \triangleright \text{nra2mq}(Q)$, where $\text{nra2mq}(Q)$ is defined recursively in Figure 8. In Figure 8, we reload the function att and assume that for an NRA query Q' , $\text{att}(Q')$ is the attribute set of the schema implied by Q' . Also, for $Q_1 \cup Q_2$ and $Q_1 \setminus Q_2$ we assume that $\text{att}(Q_i) = \{p_1, \dots, p_n\}$.

THEOREM 5.9. Let Q be a NRA query over C . Then $C \triangleright \text{nra2mq}(Q) \equiv_S Q$.

EXAMPLE 5.10. Consider the following NRA queries over $\text{rschema}(\tau_{\text{bios}})$, where fn stands for name.first , ln for

name.last, an for awards.award, and ay for awards.year:

$$Q = \pi_{fn, ln, an, ay}(\chi_{awards}(\text{bios}))$$

$$Q' = \sigma_{(rel1.ay=rel2.ay) \wedge ((rel1.fn \neq rel2.fn) \vee (rel1.ln \neq rel2.ln))}(Q \times Q)$$

Thus, Q' asks for a pair computer scientists that received an award in the same year. We illustrate some steps of $nra2mq$:

- $nra2mq(Q) = \rho_{_id, awards, birth, contribs, fn, ln} \triangleright$
 $\omega_{awards} \triangleright \rho_{_id, an, ay, birth, contribs, fn, ln} \triangleright$
 $\rho_{fn, ln, an, ay}$
- $subq_1(nra2mq(Q))$ is the sequence
 $\rho_{rel2, actRel, rel1._id, rel1.awards, rel1.birth, rel1.contribs, rel1.fn, rel1.ln} \triangleright$
 $\mu_{(actRel \neq 1) \vee ((\exists rel1.awards) \wedge (rel1.awards \neq []))} \triangleright$
 $\omega_{awards}^+ \triangleright \rho_{rel2, actRel, rel1._id, rel1.an, rel1.ay, rel1.birth, rel1.contribs, rel1.fn, rel1.ln} \triangleright$
 $\rho_{rel2, actRel, rel1.fn, rel1.ln, rel1.an, rel1.ay}$
- and $nra2mq(Q')$ is the sequence
 $pipeline(nra2mq(Q), nra2mq(Q')) \triangleright$
 $\gamma_{rel1, rel2} \triangleright \omega_{rel1} \triangleright \omega_{rel2} \triangleright$
 $\rho_{rel1.fn, rel1.ln, rel1.an, rel1.ay, rel2.fn, rel2.ln, rel2.an, rel2.ay, cond / ((rel1.ay=rel2.ay) \wedge ((rel1.fn \neq rel2.fn) \vee (rel1.ln \neq rel2.ln)))} \triangleright$
 $\mu_{(cond=true)} \triangleright$
 $\rho_{rel1.fn, rel1.ln, rel1.an, rel1.ay, rel2.fn, rel2.ln, rel2.an, rel2.ay}$ ■

Next, we consider NRA queries across several collections, and show their translation to the \mathcal{M}^{MUPGL} fragment of MQuery. Let \mathcal{S} be a set of type constraints, and Q an NRA query over the schemas for collections named C_1, \dots, C_n , with $n \geq 2$. Let us take C_1 to be the collection over which we evaluate the generated MQuery. Then, we first need to “bring in” the trees from the collections C_2, \dots, C_n , which we do in a preparatory phase $bring(C_2, \dots, C_n)$:

$$\gamma_{coll1/\epsilon} \triangleright \lambda_{coll2}^{dummy=C_2.dummy} \triangleright \dots \triangleright \lambda_{colln}^{dummy=C_n.dummy} \triangleright$$

$$\rho_{coll1, \dots, colln, actColl/[1..n]} \triangleright \omega_{actColl} \triangleright$$

$$\rho_{actColl, coll1/((actColl=i)?coll1.dummy)} \triangleright \omega_{coll1}^+ \triangleright \dots \triangleright \omega_{colln}^+$$

Second, we define a function $nra2mq^*(Q)$ that differs from $nra2mq(Q)$ in the translation of the collection names:

$$nra2mq^*(C_i) = \mu_{actColl=i} \triangleright \rho_{\{p/coll.i.p \mid p \in att(C_i)\}}$$

Finally, the translation of Q is the \mathcal{M}^{MUPGL} query $C_1 \triangleright bring(C_2, \dots, C_n) \triangleright nra2mq^*(Q)$.

THEOREM 5.11. *Let Q be an NRA query over C_1, \dots, C_n . Then $C_1 \triangleright bring(C_2, \dots, C_n) \triangleright nra2mq^*(Q) \equiv_{\mathcal{S}} Q$.*

THEOREM 5.12. \mathcal{M}^{MUPGL} captures full NRA, and \mathcal{M}^{MUPG} captures NRA over a single collection.

We observe that the above translation serves the purpose of understanding the expressive power of MQuery, but is likely to produce queries that MongoDB will not be able to efficiently execute in practice, even on relatively small database instances. We also note that the translation from NRA to MQuery works even if we allow for database instances D such that $D.C$ is not strictly of type τ_C , but may also contain other paths which are not in τ_C .

5.3 From MQuery to NRA

We show now how to translate MQuery to (recursive) NRA. First, given a set \mathcal{S} of constraints, and an MQuery stage s , we define an NRA query $mq2nra(s)$. Then, for an arbitrary MQuery $C \triangleright s_1 \triangleright \dots \triangleright s_n$, the corresponding NRA query is defined as $(mq2nra(s_1) \circ \dots \circ mq2nra(s_n))(C)$ ⁷. Below we assume that the input to $mq2nra(s)$ is a query Q with the associated attributes $att(Q)$, and τ is the type corresponding to the schema of Q . We say that a path p is *nested in* τ if $type(p', \tau) = \text{array}$ for some strict prefix p' of p . In the following, we present the translation of each stage.

MATCH. We assume match criteria φ to be in negation normal form, that is, negation appears directly in front of the atoms of the form $(p = v)$ and $\exists p$.

We first introduce the translation for conjunction- and disjunction-free criteria φ according to the type τ , that is when φ is of the form $(p = v)$, $\exists p$, $\neg(p = v)$, or $\neg \exists p$. To this purpose, we define an auxiliary function $f(\varphi)$, whose goal is to translate the criteria properly also when p is not an attribute name in $rschema(\tau)$.

For $\varphi = (p = v)$, we need to check whether p and v are “compatible” with respect to τ , that is, whether v is of type $subtree(p, \tau)$ or of type $subtree(p.0, \tau)$. When they are incompatible, we set $f(p = v) = \text{false}$; otherwise if v is of type $subtree(p, \tau)$, we define $f(p = v)$ as

- $(p = v)$, if $type(p, \tau) = \text{literal}$;
- $(p = rel_{\tau}(F))$ where $F = \{\text{attach}(p, tree(v_i))\}_{i=1}^n$, if $type(p, \tau) = \text{array}$ and $v = [v_1, \dots, v_n]$;
- $\bigwedge_{(p':v') \in rtuple_{\tau}(R_{\tau}, \epsilon, \text{attach}(p, v))} (v' \neq \text{missing}(p' = v'))$, if $type(p, \tau) = \text{object}$;

and if v is of type $subtree(p.0, \tau)$ (hence, $type(p, \tau) = \text{array}$), we define $f(p = v)$ as

- $(v \in p)$, if $type(p[], \tau) = \text{literal}$;
- $(rtuple_{\tau}(R_{\tau}, \epsilon, \text{attach}(p, v)) \in p)$, if $type(p[], \tau) = \text{object}$.

For $\varphi = \neg(p = v)$, we set $f(\varphi) = \neg(f(p = v))$.

For $\varphi = \exists p$, $f(\varphi)$ is defined as follows (we write $q \neq v$ as a shortcut for $\neg(q = v)$):

- $(p \neq \text{missing})$, if $type(p, \tau) \in \{\text{literal}, \text{array}\}$;
- $\bigvee_{p.q \in att_{\tau}(\epsilon)} (p.q \neq \text{missing})$, if $type(p, \tau) = \text{object}$.

For $\varphi = \neg \exists p$, we set $f(\varphi) = \neg(f(\exists p))$.

If the path p is not nested in τ , we translate μ_{φ} as $\sigma_{f(\varphi)}$, i.e., $mq2nra(\mu_{\varphi}) = \sigma_{f(\varphi)}$.

Consider now the case when the path p in φ is nested, and for simplicity, assume there is only one level of nesting. Further, assume that $p = q.p'$, q is a sub-relation of R_{τ} (we call it the *parent relation* of p), and p' is a prefix of some path in $att_{\tau}(q)$. Then to check the condition on p according to the semantics of match, we need to be able to access the contents of the sub-relation q by unnesting it, but to return the original (i.e., nested) relation q . So before actually doing a selection, we apply several preparatory phases.

- $AddID = \pi_{att(Q), id.att(Q)/att(Q)} \circ \nu_{id.att(Q) \rightarrow ID}$ creates an identifier for each tuple (required for negative φ , for which we need to unnest and then to nest back):

⁷We follow the convention that $(f \circ g)(x) = g(f(x))$.

- $\text{AddDup}_\varphi = \pi_{\text{att}(Q), ID, q'/q}$ creates a copy q' of the sub-relation q ;
- Prep_φ does proper preprocessing of the new attribute q'
 - $\text{Prep}_\varphi = \chi_{q'} \circ \pi_{\text{att}(Q), ID, \{a'/a \mid a \in \text{att}(q')\}}$ for positive φ ,
 - $\text{Prep}_\varphi = \chi_{q'} \circ \pi_{\text{att}(Q), ID, \text{res}/f(\varphi')} \circ \nu_{\{\text{res}\} \rightarrow \text{cond}}$ for φ of the form $\neg\varphi'$.

Then, we apply selection with the condition $f'(\varphi)$ defined as $f(\varphi_{[a \rightarrow a']})$ for positive φ and as $(\text{cond} = \{(res : \text{false})\})$ for negative φ , and finally, project away the auxiliary columns q' and ID . More precisely,

$$\text{mq2nra}(\mu_\varphi) = \text{AddID} \circ \text{AddDup}_\varphi \circ \text{Prep}_\varphi \circ \sigma_{f'(\varphi)} \circ \pi_{\text{att}(Q)}.$$

This translation can be extended to the case of multiple levels of nesting, and we omit the details, which are tedious but straightforward.

Now we deal with arbitrary criteria φ . Let $\alpha_1, \dots, \alpha_n$ be all positive literals with nested paths in φ , β_1, \dots, β_m all negative literals with nested paths in φ , and $\delta_1, \dots, \delta_k$ all literals with not nested paths in φ . For each literal about a nested path p , we need to create a separate duplicate of the parent relation q of p . So below we assume that $\text{AddDup}_{\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m}$ creates a new column named uniquely for each literal α_i and β_j , and that Prep_{β_j} projects also all these new columns and gives a unique names to the sub-relations cond (and projects them as well). We set $f'(\delta_i) = f(\delta_i)$ and let $f'(\varphi)$ be the result of replacing in φ each literal ℓ by $f'(\ell)$ (respecting the unique names of the attributes and sub-relations for each literal about a nested path). Then $\text{mq2nra}(\mu_\varphi)$ is the query

$$\begin{aligned} & \text{AddID} \circ \text{AddDup}_{\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m} \circ \\ & \text{Prep}_{\beta_1} \circ \dots \circ \text{Prep}_{\beta_m} \circ \text{Prep}_{\alpha_1} \circ \dots \circ \text{Prep}_{\alpha_n} \circ \\ & \sigma_{f'(\varphi)} \circ \pi_{\text{att}(Q)}. \end{aligned}$$

UNWIND. The unwind operator ω_p can be translated to unnest in NRA: $\text{mq2nra}(\omega_p) = \chi_p$. To deal with ω_p^+ , we first replace empty sub-relations p with $\{(a : \text{missing})\}_{a \in \text{att}_\tau(p)}$, and then apply the normal unnest. Hence, $\text{mq2nra}(\omega_p^+)$ is defined as

$$\pi_{\text{att}(Q) \setminus \{p\}, p/((p = \{\})? \{(a : \text{missing})\}_{a \in \text{att}_\tau(p)} : p)} \circ \chi_p.$$

PROJECT. A project stage ρ_P can be translated to the NRA project operator.

Let p be a path in τ . We define a function $\text{proj}(p)$. If $\text{type}(p, \tau) \in \{\text{literal}, \text{array}\}$, then $\text{proj}(p) = \{p\}$. If $\text{type}(p, \tau) = \text{object}$, then $\text{proj}(p) = \{p.p' \mid p.p' \in \text{att}_\tau(q)\}$, where q is the longest prefix of p such that $\text{type}(q, \tau) = \text{array}$, when p is nested, and $q = \epsilon$, if p is not nested.

We define the function proj also for expressions q/d . If d is a path p , we set $\text{proj}(q/p) = \{q/p\}$, if $\text{type}(p, \tau) \in \{\text{literal}, \text{array}\}$, and $\text{proj}(q/p) = \{q.r/p.r \mid p.r \in \text{proj}(p)\}$, if $\text{type}(p, \tau) = \text{object}$. If d is a (JSON) value, then simply $\text{proj}(q/d) = \{q/d\}$. For complex value definitions d , $\text{proj}(q/d)$ can be defined by analogy using $\text{proj}(p)$ when needed.

Then $\text{mq2nra}(\rho_P) = \pi_{\text{proj}(P)}$, where $\text{proj}(P) = \bigcup_{p \in P} \text{proj}(p) \cup \bigcup_{q/d \in P} \text{proj}(q/d)$.

GROUP. To translate a group operator $\gamma_{G:a_1/b_1, \dots, a_m/b_m}$, intuitively we rename attributes according to G and a_i/b_i , nest the attributes a_i into acc , and project each a_i from acc independently. For simplicity, we only show the translation when all types of paths in G and b_i 's are either literal or array. In this case, $\text{mq2nra}(\gamma_{G:a_1/b_1, \dots, a_m/b_m}) =$

$$\begin{aligned} & \pi_{\text{id}(G), a_1/b_1, \dots, a_m/b_m} \circ \nu_{\{a_1, \dots, a_m\} \rightarrow \text{acc}} \circ \\ & \pi_{\text{idAtt}(G), a_1/\text{acc}.a_1, \dots, a_m/\text{acc}.a_m} \end{aligned}$$

where $\text{id}(G) = _id.g_1/y_1, \dots, _id.g_m/y_m$ and $\text{idAtt}(G) = _id.g_1, \dots, _id.g_n$ if $G = g_1/y_1, \dots, g_m/y_m$, and $\text{id}(G) = _id/\text{null}$ and $\text{idAtt}(G) = _id$, if G is empty. This translation can be extended to the case in which some types are object by using the proj function defined above. We omit the details.

LOOKUP. For a lookup operator $\lambda_p^{p_1=C'.p_2}$, we assume that C' is of type τ' . For simplicity, we present a translation using the left join operator $R \bowtie_{\psi} R'$ that keeps the tuples in R for which it does not find matching tuples in R' , by filling in the rest of the attributes by **null**. It is possible to avoid using left join, but would make the translation less readable. To translate lookup, we first compute the left join of Q and C' according to the join condition $p_1 = C'.p_2$, then nest all attributes from C' into a sub-relation p , and finally make sure that the value of p is $\{\}$ for unmatched tuples. More precisely, we define $\text{mq2nra}(\lambda_p^{p_1=C'.p_2})(Q, C')$ as

$$\begin{aligned} & \pi_{\text{att}(Q)/\text{rel1.att}(Q), p/((p = \{\text{rel2.a:null}\}_{a \in \text{att}(C')})? \{\} : p)} \\ & (\nu_{\text{rel2.att}(C') \rightarrow p} (Q \bowtie_{Q.\text{proj}(p_1)=C'.\text{proj}(p_2)} C')) \end{aligned}$$

where $Q.\text{proj}(p_1) = C'.\text{proj}(p_2)$ is an abbreviation of a conjunction of multiple equality conditions, if $\text{subtree}(\tau, p_1)$ coincides with $\text{subtree}(\tau', p_2)$, and **false** otherwise.

We illustrate the translation of MQuery to NRA in the following examples.

EXAMPLE 5.13. Consider a collection of type τ_{bios} in Example 5.4. First, we provide the translation of some atomic criteria:

- $f(\text{name.first} = \text{"Kristen"}) = (\text{name.first} = \text{"Kristen"})$
- $f(\text{name} = \{\text{first: "Kristen"}\}) = \text{false}$ since according to τ_{bios} , the object under the key **name** should contain also the key **last**.
- $f(\text{name} = \{\text{first: "Kristen", last: "Nygaard"}\}) = ((\text{name.first} = \text{"Kristen"}) \wedge (\text{name.last} = \text{"Nygaard"}))$
- for $\varphi = (\text{contributes} = [\text{"OOP"}, \text{"Simula"}])$, $f(\varphi)$ computes a comparison between a sub-relation name and a relation value: $(\text{contributes} = \{(\text{contributes.lit} : \text{"OOP"}), (\text{contributes.lit} : \text{"Simula"})\})$
- $f(\text{contributes} = \text{"OOP"}) = (\text{"OOP"} \in \text{contributes}(\text{contributes.lit}))$

Second, we provide the translation of match stages for a

criterion about a nested path and for a complex criterion:

$$\begin{aligned}
& \text{mq2nra}(\mu_{(\text{awards.year}=2001)}) = \\
& \quad \pi_{\text{att}(\text{bios}), \text{awards1}/\text{awards}} \circ \\
& \quad \chi_{\text{awards1}} \circ \pi_{\text{att}(\text{bios}), \text{awards.award1}/\text{awards.award}, \text{awards.year1}/\text{awards.year}} \circ \\
& \quad \sigma_{(\text{awards.year1}=2001)} \circ \pi_{\text{att}(\text{bios})} \\
& \text{mq2nra}(\mu_{((\text{awards.year} \neq 1999) \vee (\text{awards.year}=2000)))} = \\
& \quad \text{AddID} \circ \pi_{\text{att}(\text{bios}), \text{ID}, \text{awards1}/\text{awards}, \text{awards2}/\text{awards}} \circ \\
& \quad \chi_{\text{awards1}} \circ \pi_{\text{att}(\text{bios}), \text{ID}, \text{awards2}, \text{res}/(\text{awards.year}=1999)} \circ \\
& \quad \nu_{\{\text{res}\} \rightarrow \text{cond}} \circ \\
& \quad \chi_{\text{awards2}} \circ \pi_{\text{att}(\text{bios}), \text{ID}, \text{cond}, \text{awards.award2}/\text{awards.award}, \text{awards.year2}/\text{awards.year}} \circ \\
& \quad \sigma_{(\text{cond}=\{(\text{res}:\text{false})\}) \vee (\text{awards.year2}=2000)} \circ \pi_{\text{att}(\text{bios})}
\end{aligned}$$

where for the first stage we omitted the creation of the identifier column. Finally, we provide the translation of a group stage:

$$\begin{aligned}
& \text{mq2nra}(\gamma_{\text{year}/\text{awards.year}; \text{persons}/\text{name}}) = \\
& \quad \pi_{\text{id.year}/\text{awards.year}, \text{persons.first}/\text{name.first}, \text{persons.last}/\text{name.last}} \circ \\
& \quad \nu_{\{\text{persons.first}, \text{persons.last}\} \rightarrow \text{acc}} \circ \\
& \quad \pi_{\text{id.year}, (\text{persons.first}, \text{persons.last})/\text{acc}.(\text{persons.first}, \text{persons.last})} \quad \blacksquare
\end{aligned}$$

It is easy to see that the translation of MQuery to NRA is of polynomial size. Although it is perhaps not surprising that MQuery can be translated to NRA, we note that it required some care to work out the details that allowed us on the one hand to correctly capture the semantics of MQuery, and on the other hand to keep the translation compact.

THEOREM 5.14. *Let F be a forest of type τ and s a stage of MQuery, then $F \triangleright s \simeq \text{mq2nra}(s)(\text{rel}_\tau(F))$ if s is not a lookup stage; otherwise $F \triangleright s[F'] \simeq \text{mq2nra}(s)(\text{rel}_\tau(F'), \text{rel}_{\tau'}(F'))$ for a forest F' of type τ' .*

PROOF. Straightforward considering the semantics of the MQuery stages and of NRA. \square

THEOREM 5.15. *Let \mathcal{S} be a set of type constraints, and q an MQuery $C \triangleright s_1 \triangleright \dots \triangleright s_m$. Then $q \equiv_{\mathcal{S}} (\text{mq2nra}(s_1) \circ \dots \circ \text{mq2nra}(s_m))(C)$.*

6. COMPLEXITY OF MQuery

In this section we report results on the complexity of different fragments of MQuery. Specifically, we are concerned with the combined and query complexity of the *Boolean query evaluation* problem, which is the problem of checking whether the answer to a given query over a given database instance is non-empty.

Our first result establishes that the full $\mathcal{M}^{\text{MUPGL}}$ and also $\mathcal{M}^{\text{MUPG}}$ are complete for exponential time with a polynomial number of alternations under LOGSPACE reductions [9, 15]. That is, have the same complexity as monad algebra with atomic equality and negation [16], which however is strictly less expressive than NRA.

THEOREM 6.1. $\mathcal{M}^{\text{MUPG}}$ and $\mathcal{M}^{\text{MUPGL}}$ are $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ -complete in combined complexity, and in AC^0 in data complexity.

PROOF SKETCH. The proof of the lower bound follows the line of the $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ -hardness proof in [16].

As for the upper bound, we provide an algorithm that follows a strategy based on starting the alternating computation from the last stage, inspired by a similar one in [16]. Let q be an $\mathcal{M}^{\text{MUPGL}}$ query and D a database instance. We check whether there is a tree in $\text{ans}_{\text{mo}}(q, D)$ using an alternating Turing machine running in exponential time with polynomially many alternations.

Intuitively, for a forest F' resulting from applying a stage s in q to a previous result F , i.e., $F' = F \triangleright s$, in general we need to check whether there is a tree and/or all trees in F' that satisfy some conditions (such as, the value of a path p in such a tree should/should not be v , or a path p should/should not exist), without explicitly constructing F' . To do so, we derive from the conditions on F' suitable conditions to be checked on F . Such conditions are obtained/guessed from the criteria in match stages, and Boolean value definitions and conditional value definitions in project stages. Both branching and alternations happen because of the group stage. For instance, if $s = \gamma_{a_1/b_1, a_2/b_2}$ and the conditions on F' contain $a_1 = []$, then we need to check that there is no tree in F satisfying $\exists b_1$. If $s = \gamma_{g/y: a_1/b_1, a_2/b_2}$ and the conditions on F' contain $\text{id}.g = v$, $a_1 \neq []$ and $a_2 \neq []$, then we need to check whether in F there is a tree satisfying $y = v$ and $\exists b_1$, and there is a tree satisfying $y = v$ and $\exists b_2$.

The overall computation starts from $F' = \text{ans}_{\text{mo}}(q, D)$, and propagates the constraints on the intermediate forests to the previous stages. The “depth” of the checks is given by the number of stages, the branching and the number of alternations are bounded by the size of q , which give us $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ upper bound.

The data complexity follows from the data complexity of NRA that is known to be in AC^0 [22]. \square

Next, we study some of the less expressive fragments of MQuery. We consider match to be an essential operator, and we start with the minimal fragment \mathcal{M}^{M} , for which we show that query answering is tractable and very efficient.

THEOREM 6.2. \mathcal{M}^{M} is LOGSPACE-complete in combined complexity.

PROOF SKETCH. The lower-bound can be shown by a reduction from the directed forest accessibility problem, known to be complete for LOGSPACE under NC^1 reducibility [11], to the problem whether $t \models \exists p$, for a tree t and a path p .

The upper-bound follows from the following facts: (i) we can check in LOGSPACE whether $t \models (p = v)$ and whether $t \models \exists p$, for a tree t , a path p , and a value v ; (ii) tree-isomorphism, needed to check equality between the subtree reached through a path p and a complex value v is in LOGSPACE [18]; (iii) the Boolean formula value problem is ALOGTIME-complete [6], and hence in LOGSPACE. \square

Next, we observe that the project and group operators allow one to create exponentially large values by duplicating the existing ones. For instance, the result of $\{\text{tree}(\{\{a:1\}\})\} \triangleright \underbrace{\rho_{a.\ell/a, a.r/a} \triangleright \dots \triangleright \rho_{a.\ell/a, a.r/a}}_{n \text{ times}}$ is a set consisting of a full binary tree of depth n . Nevertheless, without the unwind operator it is still possible to maintain tractability.

THEOREM 6.3. \mathcal{M}^{MP} is PTIME-hard in query complexity and \mathcal{M}^{MPL} is in PTIME in combined complexity.

PROOF SKETCH. The lower-bound follows from the fact that we can compute the value of a monotone Boolean circuit consisting of assignments to n variables in n project stages, and in the final match stage we can check whether the output variable evaluates to 1.

For the upper-bound, we notice that it is not necessary to materialize the exponentially large trees, instead we can work on their compact representations in the form of directed acyclic graphs (DAGs). Thus, we can devise an algorithm for which the result of each stage grows at most linearly in the size of the stage and its input set of DAGs. Hence, we can evaluate each stage on a structure that is at most polynomial. \square

We can identify the unwind operator as one of the sources of complexity, as it allows one to multiply the number of trees each time it is used in the pipeline. Indeed, adding the unwind operator alone causes already loss of tractability, provided the input tree contains multiple arrays (hence in combined complexity).

THEOREM 6.4. \mathcal{M}^{MU} is LOGSPACE-complete in query complexity and NP-complete in combined complexity.

PROOF SKETCH. For the LOGSPACE upper-bound, we observe that the number of times the unwind operation can actually multiply the number of trees is bounded by the number of arrays that are present in the input tree, and hence by a constant. Hence, we can both compute the result of the unwind stages, and evaluate the match conditions in LOGSPACE in the size of the query.

The NP lower-bound results from a straightforward encoding of the Boolean satisfiability problem: we start from an input forest containing n arrays $[0, 1]$, then we generate with n unwind stages all 2^n assignments, and finally we check with a match stage whether there is a satisfying one. The NP upper-bound follows from the next theorem. \square

Adding project and lookup does not increase the combined complexity, but does increase the query complexity, since they allow for creating multiple arrays from a fixed input tree.

THEOREM 6.5. \mathcal{M}^{MUP} and \mathcal{M}^{MUL} are NP-hard in query complexity, and \mathcal{M}^{MUL} is in NP in combined complexity.

PROOF SKETCH. The proof of the lower-bound is analogous to the one for the NP lower-bound in Theorem 6.4, except that now we can use either project or lookup to generate the forest with n arrays $[0, 1]$.

For the upper-bound, we extend the idea of using DAGs as compact representations of trees. We only specify how to evaluate an unwind stage: instead of creating a separate DAG for each element of the array, we guess an element of the array and produce at most one DAG for each input DAG. This is sufficient, since without group, we can evaluate each original tree independently of the other ones. \square

In the presence of unwind, the group operator provides another source of complexity, since in \mathcal{M}^{MUG} we can generate doubly exponential large trees, analogous to monad

algebra [16]. Let $t_0 = \text{tree}(\{_id : \{x : 0\}\})$ and $t_1 = \text{tree}(\{_id : \{x : 1\}\})$. Then the result of applying the following \mathcal{M}^{MUG} query to $\{t_0, t_1\}$ is a forest containing 2^{2^n} trees, each encoding one 2^n -bit value.

$$\left. \begin{array}{l} \gamma_{:x/_id.x} \triangleright \gamma_{x.l/x, x.r/x} \triangleright \omega_{_id.x.l} \triangleright \omega_{_id.x.r} \triangleright \\ \dots \\ \gamma_{:x/_id.x} \triangleright \gamma_{x.l/x, x.r/x} \triangleright \omega_{_id.x.l} \triangleright \omega_{_id.x.r} \end{array} \right\} n \text{ times}$$

Below we show that already \mathcal{M}^{MUG} queries are PSPACE-hard.

THEOREM 6.6. \mathcal{M}^{MUG} is PSPACE-hard in query complexity.

PROOF. Proof by reduction from the validity problem of QBF. Let φ be a quantified Boolean formula over the variables x_1, \dots, x_n of the form $Q_1 x_1 Q_2 x_2 \dots Q_n x_n. \psi$, for $Q_i \in \{\exists, \forall\}$. We construct a forest F and an \mathcal{M}^{MUG} query q such that $F \triangleright q$ is non-empty iff φ is valid.

F contains a single document d of the form $\{x: [0, 1]\}$, and q is as follows:

$$\begin{array}{l} \gamma_{x1/x, \dots, xn/x} \triangleright \omega_{_id.x1} \triangleright \dots \triangleright \omega_{_id.xn} \triangleright \mu_{\psi'} \triangleright \\ \gamma_{x1/_id.x1, \dots, x(n-1)/_id.x(n-1):val/xn} \triangleright \mu_{qua_n(val)} \triangleright \\ \dots \\ \gamma_{x1/_id.x1:val/x2} \triangleright \mu_{qua_2(val)} \triangleright \\ \gamma_{:val/x1} \triangleright \mu_{qua_1(val)} \end{array}$$

where ψ' is the criterion where each occurrence of a variable x_i in ψ is encoded by the path $_id.xi$, $qua_i(val)$ is the expression $(val = [0, 1])$ if Q_i is \forall , and the expression $(val \neq _)$ if Q_i is \exists .

The query q consists of two logical parts. In the first one we create n arrays $[0, 1]$, unwind each of them, thus creating all possible 2^n variable assignments and then filter only the satisfying ones. In the second part, for each quantifier $Q_i x_i$, we filter the assignments to the variables x_1, \dots, x_{i-1} satisfying the formula $Q_i x_i \dots Q_n x_n. \psi$ by using group. \square

7. CONCLUSIONS

Here we carried out a first formal investigation of MongoDB, a widely used NoSQL database system, with the aim of understanding its query expressiveness and complexity. We provided a formalization of the MongoDB data model, and of a core fragment, called MQuery, of the MongoDB query language. We studied the expressiveness of MQuery, showing its equivalence with NRA by developing compact translations between these two query languages. We further investigated the computational complexity of significant fragments of MQuery, obtaining several (tight) bounds in combined complexity, which range from LOGSPACE to alternating exponential-time with a polynomial number of alternations. As a byproduct, we have also established a tight complexity bound for NRA.

We briefly comment on our choice of the MongoDB query language for carrying out our investigation, as opposed to adopting a (possibly abstract and novel) language that would capture also additional features of other existing query languages for processing semi-structured or complex valued data (such as Jaql [3] or Pig Latin [19]). First of all, we see value in studying a real-world system that, although widely

adopted, still lacks a proper formalization and an understanding of its computational properties. Also, since MongoDB is still under active development, some of the insight provided by our work might help the developers to tune the system, and possibly backtrack on some choices that appear difficult to justify from a formal point of view. Turning to Jaql and Pig Latin, these are script languages, compiled into sets of map-reduce jobs for the Hadoop platform, for batch processing (note, however, that also in MongoDB the output can be stored in another collection, so it is compatible with batch processing). Moreover, Jaql is highly composable since it supports higher-order functions (all language operators can be applied to any level of nested data). This contrasts with a regular query language as the one provided by the MongoDB aggregation framework, in which the operators cannot take stage operators as parameters, and which is mostly intended to be used in an online setting. Hence, a thorough comparison with these languages would require a full investigation of their formal and computational properties, not much different in scope than what provided here, and left for future work.

We are currently working on applying the results presented here, and specifically the translation from NRA to $\mathcal{M}^{\text{MUPGL}}$, to provide high-level access to MongoDB data sources relying on the ontology-based data access paradigm [7], thus avoiding hard-coded post-processing transformations [4].

Acknowledgements. We thank Christoph Koch and Dan Suciu for helpful clarifications on nested relational algebra, and Henrik Ingo for information about MongoDB. We also thank Martin Rezk for his participation to initial work on the topic of the paper.

8. REFERENCES

- [1] Serge Abiteboul and Catriel Beeri. The power of languages for the manipulation of complex values. *Very Large Database J.*, 4(4):727–794, 1995.
- [2] Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of NoSQL languages. *SIGPLAN Notices*, 48(1):101–114, January 2013.
- [3] Kevin S. Beyer, Vuk Ercegovic, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. *Proc. of the VLDB Endowment*, 4(12):1272–1283, 2011.
- [4] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. OBDA beyond relational DBs: A study for MongoDB. In *Proc. of the 29th Int. Workshop on Description Logics (DL)*, volume 1577 of *CEUR Workshop Proceedings*, <http://ceur-ws.org/>, 2016.
- [5] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsson Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 1995.
- [6] Samuel R. Buss. The Boolean formula value problem is in ALOGTIME. In *Proc. of the 19th ACM SIGACT Symp. on Theory of Computing (STOC)*, pages 123–131. ACM Press, 1987.
- [7] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web Journal*, 8(3), 2017.
- [8] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, May 2011.
- [9] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
- [10] Latha S. Colby. A recursive algebra and query optimization for nested relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 273–283, 1989.
- [11] Stephen A Cook and Pierre McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8(3):385 – 394, 1987.
- [12] Evgeny Dantsin and Andrei Voronkov. Complexity of query answering in logic databases with complex values. In *Proc. of the 4th Int. Symp. on Logical Foundations of Computer Science (LFCS)*, pages 56–66, 1997.
- [13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [14] Stéphane Grumbach and Victor Vianu. Tractable query languages for complex object databases. In *Proc. of the 10th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*. ACM Press and Addison Wesley, 1991.
- [15] D. S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Publishers, 1990.
- [16] Christoph Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. on Database Systems*, 31(4):1215–1256, 2006.
- [17] Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, February 2010.
- [18] Steven Lindell. A LOGSPACE algorithm for tree canonization (extended abstract). In *Proc. of the 24th ACM SIGACT Symp. on Theory of Computing (STOC)*, pages 400–404, 1992.
- [19] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1099–1110, 2008.
- [20] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proc. of the 25th Int. World Wide Web Conf. (WWW)*, pages 263–273, 2016.
- [21] Michael Stonebraker and Ugur Cetintemel. “one size fits all”: An idea whose time has come and gone. In *Proc. of the 21st IEEE Int. Conf. on Data Engineering*

(*ICDE*), pages 2–11, 2005.

- [22] Dan Suciu and Val Tannen. A query language for NC. *Journal of Computer and System Sciences*, 55(2):299–321, 1997.
- [23] Stan J Thomas and Patrick C Fischer. Nested relational structures. *Advances in Computing Research*, 3:269–307, 1986.
- [24] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. of the VLDB Endowment*, 2(2):1626–1629, 2009.
- [25] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1):363–377, 2001.
- [26] Jan Van den Bussche and Jan Paredaens. The expressive power of complex values in object-based data models. *Information and Computation*, 120(2):220–236, 1995.

APPENDIX

A. EXAMPLES OF MongoDB QUERIES

MongoDB provides two main query mechanisms. The basic form of query is a *find* query, which allows one to filter out documents according to some (Boolean) criteria and to return, for each document passing the filter, a tree containing a subset of the key-value pairs in the document. Specifically, a find query has two components, where the first one is a *criterion* for selecting documents, and the second one is a *projection condition*.

EXAMPLE A.1. The following MongoDB find query selects from the `bios` collection the documents talking about scientists whose first name is Kristen, and for each document only returns the full name and the date of birth.

```
db.bios.find(
  {"name.first": {$eq: "Kristen"}},
  {"name" : 1, "birth" : 1}
)
```

When applied to the document in Figure 1, it returns the following tree:

```
{
  "_id": 4,
  "birth": "1926-08-27",
  "name": {
    "first": "Kristen", "last": "Nygaard" }
}
```

Observe that by default the document identifier is included in the answer of the query. \square

Note that with a find query we can either obtain the original documents as they are, or we can modify them by specifying in the projection condition only a subset of the keys, thus retaining in the answer only the corresponding key-value pairs. However, we cannot change the shape of the individual pairs.

A more powerful querying mechanism is provided by the *aggregation framework*, in which a query consists of a pipeline of *stages*, each transforming a forest into a new forest. We call this transformation pipeline an *MQuery*. One of the main differences with find queries is that MQuery can manipulate the shape of the trees.

EXAMPLE A.2. The following MQuery essentially does the same as the previous find query, but now it flattens the complex object name into two key-value pairs.

```
db.bios.aggregate([
  {$match: {"name.first": {$eq: "Kristen"}}},
  {$project: {
    "birth": true, "firstName": "$name.first", "lastName": "$name.last" } }
])
```

So the document from our running example will be transformed into the following tree:

```
{
  "_id" : 4,
  "birth": "1926-08-27",
  "firstName": "Kristen",
  "lastName": "Nygaard"
}
```

\square

EXAMPLE A.3. Consider the MQuery

```
db.bios.aggregate([
  {$project: { "name": true,
    "award1": "$awards", "award2": "$awards" } },
  {$unwind: "$award1"},
  {$unwind: "$award2"},
  {$project: {
    "name": true, "award1": true, "award2": true,
    "twoInOneYear": { $and: [
      {$eq: ["$award1.year", "$award2.year"]},
      {$ne: ["$award1.award", "$award2.award"]} ] } },
  {$match: { "twoInOneYear": true } },
  {$project: { "firstName": "$name.first",
    "lastName": "$name.last",
    "awardName1": "$award1.award",
    "awardName2": "$award2.award",
    "year": "$award1.year" } },
])
```

It consists of 6 stages and retrieves all persons who received two awards in one year. The first stage keeps the complex object `name`, creates two copies of the array `awards`, and projects away all other fields. The second and third stages flatten (unwind) the two copies (`award1` and `award2`) of the array of awards (which intuitively creates a cross-product). The fourth step compares awards pairwise and creates a new key (`twoInOneYear`) whose value is true if the scientist has two awards in one year. The fifth one selects the documents of interests (those where `twoInOneYear` is true), and the final stage renames and projects keys.

By applying the query to the document in Figure 1, we obtain:

```
{
  "_id": 4,
  "firstName": "Kristen",
  "lastName": "Nygaard",
  "awardName1": "IEEE John von Neumann Medal",
  "awardName2": "Turing Award",
  "year": 2001
}
```

□

We note that the unwind operator creates a new document for every element in the array. Thus, unwinding `awards` (once) in the document in our running example will output 3 documents, only one of which satisfies the subsequent selection stages. In the example below we illustrate the group stage, which combines different documents into one.

EXAMPLE A.4. The following query returns for each year all scientists that received an award in that year.

```
db.bios.aggregate([
  { $unwind: "$awards" },
  { $group: {
    _id: { "year": "$awards.year" }, "names": { $addToSet: "$name" } } },
])
```

Running this query over the database consisting of the document in Figure 1, produces the following output:

```
{
  "_id": { "year": 2001 },
  "names": [
    { "first": "Kristen", "last": "Nygaard" } ]
},
{
  "_id": { "year": 1999 },
  "names": [
    { "first": "Kristen", "last": "Nygaard" } ]
}
```

□

B. TREE OPERATIONS

In the following, let $t = (N, E, L_n, L_e)$ be a tree. Below, when we mention reachability, we mean reachability along the edge relation.

subtree the subtree of t rooted at x and induced by M , for $n \in M$ and $M \subseteq N$, denoted $\text{subtree}(t, x, M)$, is defined as $(N', E|_{N' \times N'}, L_n|_{N'}, L_e|_{E'})$ where N' is the subset of nodes in M reachable from x through nodes in M . We write $\text{subtree}(t, M)$ as abbreviation for $\text{subtree}(t, \text{root}(t), M)$.

For a path p with $|\llbracket p \rrbracket^t| = 1$, the subtree $\text{subtree}(t, p)$ of t hanging from p is defined as $\text{subtree}(t, r_p, N')$ where $\{r_p\} = \llbracket p \rrbracket^t$, and N' are the nodes reachable from r_p via E . For a path p with $|\llbracket p \rrbracket^t| = 0$, $\text{subtree}(t, p)$ is defined as $\text{tree}(\text{null})$.

attach The tree $\text{attach}(k_1 \dots k_n, t)$ constructed by inserting the path $k_1 \dots k_n$ on top of the tree t , for $n \geq 1$, is defined as (N', E', L'_n, L'_e) , where

- $N' = N \cup \{x_0, x_1, \dots, x_{n-1}\}$, for fresh x_0, \dots, x_{n-1} .
- $E' = E \cup \{(x_0, x_1), (x_1, x_2), \dots, (x_{n-1}, \text{root}(t))\}$,
- $L'_n = L_n \cup \{(x_0, \llbracket \cdot \rrbracket), \dots, (x_{n-1}, \llbracket \cdot \rrbracket)\}$,
- $L'_e = L_e \cup \{((x_0, x_1), k_1), \dots, ((x_{n-2}, x_{n-1}), k_{n-1}), ((x_{n-1}, \text{root}(t)), k_n)\}$.

intersection Let t_1 and t_2 be trees. The function $t_1 \cap t_2$ returns the set of pairs of nodes $(x_n, y_n) \in N^1 \times N^2$ reachable along identical paths in t_1 and t_2 , that is, such that there exist $(x_0, x_1), \dots, (x_{n-1}, x_n)$ in E^1 , for $x_0 = \text{root}(t_1)$, and $(y_0, y_1), \dots, (y_{n-1}, y_n)$ in E^2 , for $y_0 = \text{root}(t_2)$, with $L_N^1(x_i) = L_N^2(y_i)$ and $L_E^1(x_{i-1}, x_i) = L_E^2(y_{i-1}, y_i)$, for $1 \leq i \leq n$.

merge Let t_1, t_2 be trees (N^j, E^j, L_n^j, L_e^j) , $j = 1, 2$, such that $N^1 \cap N^2 = \emptyset$, and for each path p leading to a leaf in t_2 , i.e., $t_2 \models (p = v)$ for some literal value v , we have that $t_1 \not\models \exists p$ and the other way around. Then the tree $t_1 \oplus t_2$ resulting from merging t_1 and t_2 is defined as (N, E, L_n, L_e) , where

- $N = N^1 \cup N^{2'}$, for $N^{2'} = N^2 \setminus \{x_2 \mid (x_1, x_2) \in t_1 \cap t_2\}$
- $E = E^1 \cup (E^2 \cap (N^{2'} \times N^{2'})) \cup ((t_1 \cap t_2) \circ E^2)$
- $L_n = L_n^1 \cup L_n^2|_{N^{2'}}$
- $L_e = L_e^1 \cup L_e^2|_{N^{2'} \times N^{2'}} \cup \{(x_1, y_2), \ell \mid L_e^2(y_1, y_2) = \ell, (x_1, y_1) \in t_1 \cap t_2\}$

minus $t_1 \setminus t_2$ is subtree(t_1, N') where $N' = N_1 \setminus N_2$.

array Let $\{t_1, \dots, t_n\}$, $n \geq 0$, be a forest and p a path. The operator $\text{array}(\{t_1, \dots, t_n\}, p)$ creates the tree encoding the array of the values of the path p in the trees t_1, \dots, t_n . Let $t_j^p = \text{subtree}(t_j, p)$ with (N^j, E^j, L_n^j, L_e^j) where all N^j are mutually disjoint, and $r_j = \text{root}(t_j^p)$. Then, $\text{array}(\{t_1, \dots, t_n\}, p)$ is the tree (N, E, L_n, L_e) where

- $N = \left(\bigcup_{j=1}^n N^j\right) \cup \{v_0\}$,
- $E = \left(\bigcup_{j=1}^n E^j\right) \cup \{(v_0, r_1), \dots, (v_0, r_n)\}$,
- $L_n = \left(\bigcup_{j=1}^n L_n^j\right) \cup \{(v_0, [\cdot])\}$,
- $L_e = \left(\bigcup_{j=1}^n L_e^j\right) \cup \{((v_0, r_1), 0), \dots, ((v_0, r_n), n-1)\}$.

We also define $\text{subtree}(t, p)$ for paths p such that $\|p\|^t > 1$. In this case it returns the tree encoding the array of all subtrees hanging from p . Formally, $\text{subtree}(t, p) = \text{array}(\{t_1, \dots, t_n\}, \varepsilon)$, where $\{r_1, \dots, r_n\} = \|p\|^t$, N_j the set of nodes reachable from r_j via E , and $t_j = \text{subtree}(t, r_j, N_j)$. We observe that the definition of the array operator is recursive as it uses the generalized subtree operator.

B.1 Expressing $\exists p$ in value definitions

We can express $\exists p$ as $(p? \text{true} : ((\neg(p = \text{null}) \wedge \neg(p = \text{false}) \wedge \neg(p = 0)) ? \text{true} : \text{false}))$.

C. NESTED RELATIONAL ALGEBRA TO MQuery

LEMMA C.1. *The result of pipeline(q_1, q_2) contains the result of q_i in the trees with $\text{actRel} = i$ under the key $\text{rel}i$.*

PROOF. Let F be a forest, and F_0 the result of evaluating of the first 3 stages in pipeline(q_1, q_2) over F . Then F_0 satisfies the property:

(\star) for each tree t in F_0 , if $t \models (\text{actRel} = 1)$, then $t \models \exists \text{rel}1 \wedge \neg \exists \text{rel}2$, and if $t \models (\text{actRel} = 2)$, then $t \models \exists \text{rel}2 \wedge \neg \exists \text{rel}1$.

Moreover, for each tree $t \in F$, there are exactly two trees t_1 and t_2 in F_0 such that $t_1 \models (\text{actRel} = 1)$, subtree($t_1, \text{rel}1$) coincides with t , and $t_2 \models (\text{actRel} = 2)$, subtree($t_2, \text{rel}2$) coincides with t . These follow from the semantics of conditional value definition and of $\rho_{p/q}$ when q is missing from the input trees.

Let $F_1 = F_0 \triangleright \text{subq}_1(q_1)$. We prove that

(clean) F_1 satisfies (\star),

(own) $(F_1 \triangleright \mu_{\text{actRel}=1})$, coincides with $F \triangleright q_1$, and

(other) $(F_1 \triangleright \mu_{\text{actRel}=2})$ coincides with $(F_0 \triangleright \mu_{\text{actRel}=2})$, which coincides with F (i.e., the “other” trees are not affected).

It is sufficient to prove the above for the case of q_1 being a single stage pipeline s . Consider the following cases:

- s is a match stage μ_φ . Then $\text{subq}_1(q_1) = \mu_{(\text{actRel} \neq 1) \vee \varphi_{[p/\text{rel}1.p]}}$. Since match does not alter the structure of the trees, F_1 satisfies (\star).

Let $t \in (F_0 \triangleright \mu_{(\text{actRel} \neq 1) \vee \varphi_{[p/\text{rel}1.p]}} \triangleright \mu_{(\text{actRel}=1)})$. Then by the properties of match, it follows that $t \in (F_0 \triangleright \mu_{(\text{actRel}=1)} \triangleright \mu_{\varphi_{[p/\text{rel}1.p]}})$. By assumption, $(F_0 \triangleright \mu_{(\text{actRel}=1)})$ coincides with F , therefore we obtain that t is in $F \triangleright q_1$ (up to proper renaming). Similarly, in the other direction, when $t \in (F \triangleright q_1)$, we derive that $t \in (F_1 \triangleright \mu_{(\text{actRel}=1)})$.

Since the query $\mu_{(\text{actRel} \neq 1) \vee \varphi_{[p/\text{rel}1.p]}} \triangleright \mu_{(\text{actRel}=2)}$ is equivalent to the query $\mu_{(\text{actRel}=2)}$, we obtain that the forest $(F_0 \triangleright \mu_{(\text{actRel} \neq 1) \vee \varphi_{[p/\text{rel}1.p]}} \triangleright \mu_{(\text{actRel}=2)})$ coincides with $(F_0 \triangleright \mu_{(\text{actRel}=2)})$.

- s is an unwind stage ω_p^+ . Then $\text{subq}_1(q_1) = \omega_{\text{rel}1.p}^+$. First, $\text{subq}_1(q_1)$ does not affect the trees with $\text{actRel} = 2$ because there does not exist the path $\text{docl}.p$, and $\text{subq}_1(q_1)$ will preserve all such trees as they are. Second, the trees that contain the path $\text{docl}.p$ (hence, with $\text{actRel} = 1$), will be affected in exactly the same way as the trees in F would be affected by q_1 . Finally, since unwind does not affect other paths than p , we have that F_1 satisfies the clean specialization property.
- s is an unwind stage ω_p . Then $\text{subq}_1(q_1) = \mu_{(\text{actRel} \neq 1) \vee ((\exists \text{rel}1.p) \wedge (\text{rel}1.p \neq []))} \triangleright \omega_{\text{rel}1.p}^+$. Again, $\text{subq}_1(q_1)$ does not affect the trees with $\text{actRel} = 2$ because they will all pass the match stage and the subsequent unwind will preserve them as they are. Second, we note that evaluating q_1 over F will remove trees where path p does not exist, or p exists and its value is **null**, or empty array. This is done by $\text{subq}_1(q_1)$ in the match stage. The subsequent unwind acts as the unwind above. Again, we have that F_1 satisfies the clean specialization property.

- s is a project stage $\rho_{p,q/d}$. Then, $\text{subq}_1(q_1) = \rho_{\text{rel2}, \text{actRel}, \text{rel1.}_id, \text{rel1.p}, \text{rel1.q}=(\text{actRel}=1)/d_{[p'/\text{rel1.p}]/\text{dummy}}}$. It is easy to see that (clean) and (other) are satisfied. As for (own), the trees with $\text{actRel} = 1$ will keep the paths rel1._id , rel1.q and the value of the path rel1.p will be defined by d . Hence, (own) also holds.
- s is a group stage $\gamma_{g/y;a/b}$. Then $\text{subq}_1(q_1) = \gamma_{\text{rel1.g}/\text{rel1.y}, \text{actRel:rel1.a}/\text{rel1.b}, \text{rel2} \triangleright$

$$\begin{aligned} & \rho_{\text{rel2}, \text{actRel}/_id.\text{actRel}, \text{rel1.a}, \text{rel1.}_id.\text{g}/_id.\text{rel1.g}} \triangleright \\ & \rho_{\text{actRel}, \{\text{rel1}=(\text{actRel}=i)/\text{rel1}/\text{dummy}\}_{i=1,2}} \triangleright \\ & \omega_{\text{rel2}}^+ \end{aligned}$$

The result of the first stage is $n + 1$ trees where

- one tree originates from all trees with $\text{actRel} = 2$, the value of rel2 is the array of all such rel2 and rel1.a is an empty array.
- n is the number of different values v_1, \dots, v_n of $\text{rel1.g}'$ in all trees with $\text{actRel} = 1$, and each of the n trees originates from a subset of the trees with $\text{actRel} = 1$ and $\text{rel1.g}' = v_i$, the value of rel2 is the empty array, the value of rel1.a is all $\text{rel1.a}'$ in this subset of trees, and the value of rel1.g is v_i .

The result of the second stage is $n + 1$ trees where some paths in $_id$ are renamed. The result of the third stage is a forest satisfying the clean specialization property. In the forth stage, the array rel2 is unwinded, hence the trees with $\text{actRel} = 2$ are brought in the original shape. It is easy to see that all properties are satisfied.

Since the translation is symmetric, we have also that $F_2 = F_1 \triangleright \text{subq}_2(q_2)$ satisfies the corresponding properties (clean), (own) and (other). \square

THEOREM 5.9. *Let Q be a NRA query over C . Then $C \triangleright \text{nra2mq}(Q) \equiv_S Q$.*

PROOF. Follows from the definition of $\text{rschema}_\tau(C)$, Lemma C.1 and the semantics of MQuery stages. \square

D. COMPLEXITY OF MQUERY

THEOREM 6.1. $\mathcal{M}^{\text{MUPG}}$ and $\mathcal{M}^{\text{MUGL}}$ are $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ -hard and in EXSPACE in combined complexity and in AC^0 in data complexity.

LEMMA D.1. $\mathcal{M}^{\text{MUPG}}$ is $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ -hard in combined complexity.

PROOF. We adapt the proof of $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ -hardness from [16].

Let $M = (\Sigma, Q, \delta, q_0, F)$ be an alternating Turing machine that runs in time $2^{p_1(n)}$ with $p_2(n)$ alternations on inputs of size n , where Σ the tape alphabet, Q is the set of states partitioned into existential Q_\exists and universal Q_\forall states, $\delta : Q \times \Sigma \times \{1, 2\} \rightarrow Q \times \Sigma \times \{-1, 0, +1\}$ the transition function, which for a state q and symbol s gives two instructions $\delta(q, s, 1)$ and $\delta(q, s, 2)$, q_0 the initial state and $F \subseteq Q$ the set of accepting states.

Following Koch, we simulate the computation of M in $\mathcal{M}^{\text{MUPG}}$. Each run of M is a tree of configurations of depth bounded by $p_2(n) \cdot 2^{p_1(n)}$, and each configuration consists of a tape of length bounded by $2^{p_1(n)}$, a current state and a position marker on the tape. We construct an $\mathcal{M}^{\text{MUPG}}$ q and a forest F such that $F \triangleright q$ is non-empty iff M accepts its input. F consists of a single document containing the key-value pair $_id: 1$.

- The tape of a configuration is modeled as a nested object of nested depth $p_1(n)$ and with $2^{p_1(n)}$ leaves. The position of the head on the tape is represented by an extended tape alphabet $\Sigma' = \Sigma \cup \{\bar{s} \mid s \in \Sigma\}$. That is, the symbol \bar{s} in a tape cell indicates that the cell stores symbol s and it is the current position of the head. The following is a valid tape:

$$\text{"tape": } \{ \text{"1": } \{ \text{"1": } \text{"0"}, \text{"r": } \text{"0"} \}, \text{"r": } \{ \text{"1": } \text{"#"}, \text{"r": } \text{"#"} \} \}.$$

We can compute the set of all $m^{2^{p_1(n)}}$ tapes (including non-valid ones) by the query:

$$\begin{aligned} \text{Tapes} = & \rho_{\text{tape}/[\Sigma']} \triangleright \rho_{l/\text{tape}, r/\text{tape}} \triangleright \omega_l \triangleright \omega_r \triangleright \rho_{\text{tape.l}/l, \text{tape.r}/r} \triangleright \gamma_{\text{tape}} \triangleright \\ & \dots \\ & \rho_{l/\text{tape}, r/\text{tape}} \triangleright \omega_l \triangleright \omega_r \triangleright \rho_{\text{tape.l}/l, \text{tape.r}/r} \triangleright \gamma_{\text{tape}} \end{aligned} \left. \vphantom{\begin{aligned} \text{Tapes} = \end{aligned}} \right\} p_1(n) \text{ times}$$

Where for a set S , the value definition $[S]$ means a constant array consisting of all elements in S (we view everything as strings). The result of this query (on F) is a single document containing an array of all possible tapes under the key tape .

- In turn, a configuration is a pair, consisting of a tape and a state. We can compute all possible (including non-valid ones) configurations by the following $\mathcal{M}^{\text{MUPG}}$:

$$\text{Configs} = \text{Tapes} \triangleright \rho_{\text{tape}, \text{state}/[Q]} \triangleright \omega_{\text{tape}} \triangleright \omega_{\text{state}} \triangleright \rho_{c.\text{tape}/\text{tape}, c.\text{state}/\text{state}}$$

The result of Configs is a set of trees, each containing one possible configuration under the key c .

- Next, we are going to construct a query that computes the pairs of configurations c_1 and c_2 such that c_2 is a possible immediate successor of c_1 according to δ (also including pairs of non-valid configurations). First, we create all pairs of configurations c_1 and c_2 , and make working copies w_1 and w_2 of the tapes.

$$\text{Prepare-succ} = \text{Configs} \triangleright \gamma_{:c1/c, c2/c} \triangleright \omega_{c1} \triangleright \omega_{c2} \triangleright \rho_{\text{succ}/\{c1/c1, c2/c2\}, w1/c1.\text{tape}, w2/c2.\text{tape}}$$

Second, to check that c_1 is a possible successor of c_2 , we verify that w_1 and w_2 differ at at most two consecutive tape positions. The tapes are of exponential length, but we can find these two positions by doing a number of checks that is equal to the depth of the value encoding a tape minus 1. Namely, we iteratively compare the halves of the working copies, and in the next step the working copies become the halves which are not equal (see [16] for more details):

- If $w_1.l = w_2.l$ (the left halves of the tapes are equal), we replace w_1 by $w_1.r$ and w_2 by $w_2.r$
- If $w_1.r = w_2.r$ (the right halves of the tapes are equal), we replace w_1 by $w_1.l$ and w_2 by $w_2.l$
- If $w_1.ll = w_2.ll$ and $w_1.r.r = w_2.r.r$ (the left and the right quarters of the tapes are equal, so the difference should be in the “inner” part of the tree), we replace $w_1.l$ by $w_1.l.r$, $w_1.r$ by $w_1.r.l$ and $w_2.l$ by $w_2.l.r$, $w_2.r$ by $w_2.r.l$

We implement zooming-in by the query:

$$\begin{aligned} \text{Zoom-in} = & \rho_{\text{succ}, w1/((w1.l=w2.l)?w1.r:((w1.r=w2.r)?w1.l:(((w1.l.l=w2.l.l)\wedge(w1.r.r=w2.r.r))?\{l/w1.l.r,r/w1.r.l\}:\text{null})))} \\ & w2/((w1.l=w2.l)?w2.r:((w1.r=w2.r)?w2.l:(((w1.l.l=w2.l.l)\wedge(w1.r.r=w2.r.r))?\{l/w2.l.r,r/w2.r.l\}:\text{null}))) \\ & \mu_{\neg(w1=\text{null})} \end{aligned}$$

After finding the two positions where the tapes differ, we check that the head is over one of these positions.

$$\text{Head} = \mu_{\bigvee_{s \in \Sigma} ((w1.l=\bar{s}) \vee (w1.r=\bar{s}))}$$

Then, we check that the difference is according to the transition function δ . Let criterion φ_δ be the disjunction of the following formulas $\varphi_{q,s,q',z,l}$, for each instruction $\delta(q, s, i) = (q', z, l)$:

$$\begin{aligned} \varphi_{q,s,q',z,0} = & (\text{succ}.c1.\text{state} = q) \wedge (\text{succ}.c2.\text{state} = q') \wedge ((w1.l = \bar{s}) \wedge (w2.l = \bar{z}) \wedge \bigvee_{b \in \Sigma} ((w1.r = b) \wedge (w2.r = b)) \vee \\ & (w1.r = \bar{s}) \wedge (w2.r = \bar{z}) \wedge \bigvee_{b \in \Sigma} ((w1.l = b) \wedge (w2.l = b))) \\ \varphi_{q,s,q',z,+1} = & (\text{succ}.c1.\text{state} = q) \wedge (\text{succ}.c2.\text{state} = q') \wedge (w1.l = \bar{s}) \wedge (w2.l = z) \wedge \bigvee_{b \in \Sigma} ((w1.r = b) \wedge (w2.r = \bar{b})) \\ \varphi_{q,s,q',z,-1} = & (\text{succ}.c1.\text{state} = q) \wedge (\text{succ}.c2.\text{state} = q') \wedge (w1.r = \bar{s}) \wedge (w2.r = z) \wedge \bigvee_{b \in \Sigma} ((w1.l = b) \wedge (w2.l = \bar{b})) \end{aligned}$$

Finally, the query Succ that computes pairs of successor configurations is:

$$\text{Succ} = \text{Prepare-succ} \triangleright \underbrace{\text{Zoom-in} \triangleright \dots \triangleright \text{Zoom-in}}_{p_1(n)-1 \text{ times}} \triangleright \text{Head} \triangleright \mu_{\varphi_\delta} \triangleright \rho_{\text{succ}}$$

- To encode alternations, we first need to compute computation paths of length up to $2^{p_1(n)}$ that we represent by pairs (c_1, c_2) : c_2 is reachable from c_1 in at most $2^{p_1(n)}$ steps, moreover if the state of c_1 is existential, then each of the intermediate configurations before reaching c_2 must be existential, and likewise if the state of c_1 is universal. We implement “at most” by means of the “stay transitions” (c, c) added to Succ. We compute these computation paths iteratively:

$$\text{CP}_0 = \text{Succ}$$

$$\text{CP}_{i+1} = \text{CP}_i \triangleright \gamma_{:s1/\text{succ}, s2/\text{succ}} \triangleright \omega_{s1} \triangleright \omega_{s2} \triangleright \mu_{s1.c2=s2.c1} \triangleright \mu_{(s1.c1.\text{state} \in Q_\exists) \leftrightarrow (s2.c1.\text{state} \in Q_\exists)} \triangleright \rho_{\text{succ}/\{c1/s1.c1, c2/s2.c2\}}$$

where $(p \in A)$, for a set A , is a shortcut for $\bigvee_{a \in A} (p = a)$, and $\varphi_1 \leftrightarrow \varphi_2$ is a shortcut for $(\neg \varphi_1 \vee \varphi_2) \wedge (\neg \varphi_2 \vee \varphi_1)$.

We can now compute the sets A_i of configurations that lead to an accepting state in i alternations:

$$\begin{aligned} A_1 = & \text{CP}_{p_1(n)} \triangleright \gamma_{\text{cp}/\text{succ}} \triangleright \rho_{\text{cp}, s/\text{cp}} \triangleright \omega_s \triangleright \mu_{(s.c2.\text{state} \in F) \wedge (s.c1.\text{state} \in Q_\exists)} \triangleright \rho_{\text{cp}, a/s.c1} \triangleright \gamma_{\text{cp}:a} \triangleright \rho_{\text{cp}/_id.\text{cp}, a} \\ A_{i+1} = & A_i \triangleright \rho_{\text{cp}, a, s/\text{cp}} \triangleright \omega_s \triangleright \omega_a \triangleright \rho_{\text{cp}, s, \text{inAi}/(s.c2=a)} \triangleright \gamma_{\text{cp}, s:\text{inAi}} \triangleright \mu_{\text{inAi}=[\text{false}]} \triangleright \rho_{\text{cp}/_id.\text{cp}, s/_id.s} \triangleright \\ & \mu_{(s.c1.\text{state} \in Q_\exists) \leftrightarrow (s.c1.\text{state} \in Q_\exists)} \triangleright \rho_{\text{cp}, a/s.c1} \triangleright \gamma_{\text{cp}:a} \triangleright \rho_{\text{cp}/_id.\text{cp}, a} \end{aligned}$$

- Finally, we check that the initial computation is in $A_{p_2(n)}$. The initial configuration has a tape, where the input string w of length n is padded with $2^{p_1(n)} - n$ #-symbols. Let v_w be the nested value of depth $\lceil \log_2 n \rceil$ representing w padded with $2^{\lceil \log_2 n \rceil} - n$ #-symbols (it can be computed in LOGSPACE). Then the initial configuration can be computed, and checked whether in $A_{p_2(n)}$ as follows:

$$\begin{aligned} C_0 \text{in} A_{p_2(n)} = & A_{p_2(n)} \triangleright \rho_{a, \text{tape}.l/v_w, \text{tape}.r/\text{"\#"}} \triangleright \underbrace{\rho_{a, \text{tape}.l, \text{tape}.r/\{l/\text{tape}.r, r/\text{tape}.r\}} \triangleright \dots \triangleright \rho_{a, \text{tape}.l, \text{tape}.r/\{l/\text{tape}.r, r/\text{tape}.r\}}}_{\lceil \log_2 n \rceil \text{ times}} \\ & \underbrace{\rho_{a, \text{tape}.l/\text{tape}, \text{tape}.r/\{l/\text{tape}.r, r/\text{tape}.r\}} \triangleright \dots \triangleright \rho_{a, \text{tape}.l/\text{tape}, \text{tape}.r/\{l/\text{tape}.r, r/\text{tape}.r\}} \triangleright \rho_{a, c0.\text{tape}/\text{tape}, c0.\text{state}/\text{"q0"}}}_{p_1(n) - \lceil \log_2 n \rceil - 1 \text{ times}} \\ & \omega_a \triangleright \mu_{a=c0} \end{aligned}$$

(where in project $\text{tape.r}/\{l/\text{tape.r}, r/\text{tape.r}\}$ can be seen as a shortcut for $\text{tape.r.l}/\text{tape.r}$, $\text{tape.r.r}/\text{tape.r}$. In fact, it is possible to write such a value definition in MongoDB. We will use this syntax for brevity also in what follows.) The part of the query that computes the initial configuration takes v_w , pads it with $\#$ -symbols so as to have tape having the value of depth $\lceil \log_2 n \rceil + 1$ where tape.r consists entirely of $\#$'s. Then in the second line it increases the depth of the tape value to $p_1(n)$ by iteratively assigning the previous value of tape to tape.l and duplicating the value of tape.r to tape.r.l and tape.r.r . See [16] for more details.

Thus we obtain that $\{\text{tree}(\{_id : 1\})\} \triangleright C_0 \text{in } A_{p_2(n)}$ is non-empty iff M accepts w . \square

LEMMA D.2. $\mathcal{M}^{\text{MUPGL}}$ is in $\text{TA}[2^{n^{O(1)}}, n^{O(1)}]$ in combined complexity.

PROOF. Let $q = C \triangleright s_1 \triangleright \dots \triangleright s_n$ be an $\mathcal{M}^{\text{MUPGL}}$ query, and D a MongoDB instance. We provide an algorithm to check that $\text{ans}_{\text{mo}}(q, D)$ is non-empty.

We assume that q is of the following form:

- we consider atoms of the form $p = v$, $p \neq v$, $p = p$, $p \neq p$, $\exists p$, $\neg \exists p$, and assume that the criteria in match stages, Boolean value definitions and conditions in conditional value definitions are monotone Boolean expressions over such atoms (i.e., use only conjunction and disjunction). To make sure that the atomic expressions in such value definitions are of this form, we replace atomic expressions p , for a path p , with $\exists p \wedge (p \neq \text{null}) \wedge (p \neq \text{false}) \wedge (p \neq 0)$, and get rid of the “constant” expressions of the form v for a value v , and $[d_1, \dots, d_k]$ as follows. First, replace the former with **true** when $v \neq \text{null}$, $v \neq \text{false}$ and $v \neq 0$, and with **false** otherwise, and replace the latter by **true**. Second, simplify the Boolean combinations with **true** and **false** in the standard way (e.g., $\text{true} \vee e$ becomes **true**, and $\text{true} \wedge e$ becomes e), and simplify conditional value definitions that have **true** or **false** as the condition in the obvious way.
- each project stage is of the form $\rho_{p_1, \dots, p_n, q/d}^{ni}$, that is, contains at most one projection element defining the value of a path q . We can achieve it for an arbitrary project stage $\rho_{p_1, \dots, p_n, q_1/d_1, \dots, q_m/d_m}^{ni}$ by splitting it into m project stages: $\rho_{p_1, \dots, p_n, q_1/d_1}^{ni} \triangleright \rho_{p_1, \dots, p_n, q_2/d_2}^{ni} \triangleright \dots \triangleright \rho_{p_1, \dots, p_n, q_{m-1}/d_{m-1}, q_m/d_m}^{ni}$.

Let $\text{ans}_{\text{mo}}(q, D) = F_n$. The algorithm is to check whether there is a tree in F_n . We do it recursively as follows. Assume that $F \triangleright s = F'$ and we want check whether a tree satisfying a set ψ of atoms (of the considered form) is in F' . Then, the check amounts to the following:

- if $s = \mu_\varphi$, then we guess atoms e_1, \dots, e_m appearing in φ so that assigning them the true value makes φ true, add to ψ the conditions e_1, \dots, e_m . If the new conditions ψ' are consistent, we check whether there is a tree satisfying ψ' in F . Otherwise, we report a failure.
- if $s = \omega_p$, then we replace p by $p.i$ in all conditions in ψ about p and check whether there is a tree with the new conditions ψ' in F .
- if $s = \omega_p^+$, then we guess whether with or without index, and in the former case we replace p by $p.i$ in all conditions in ψ about p , in the latter ψ is not changed. Then we check whether there is a tree with the new conditions ψ' in F .
- if $s = \rho_{p_1, \dots, p_l, q/d}$, we do not do anything for p_i . As for q/d , we remove $\exists q$ if it is in ψ , and proceed as follows:
 1. if d is a path q' , we replace each occurrence of q in ψ by q' .
 2. if d is a constant non-array value,
 - (a) if there is a condition of the form $q = v$ in ψ , we remove it from ψ , check whether $v = d$, and if not, we report failure.
 - (b) if there is a condition of the form $q \neq v$ in ψ , we remove it from ψ , check whether $v \neq d$, and if not, we report failure.
 - (c) if there is a condition of the form $p' = v'$ in ψ , for a prefix p' of q , we extract the value v for p (it will be the value of the subtree in v reachable by path q' such that $p'.q' = q$) if it is possible and proceed as in (a), otherwise we report a failure.
 - (d) if there is a condition of the form $q.p' = v'$ or $q.p' \neq v'$ in ψ , we extract the value definition d' from d reachable by path p' if it is possible and proceed as in the case $q.p'/d'$, otherwise we report a failure.
 - (e) if there is a condition of the form $q.i = v'$ in ψ , we report a failure.
 3. if d is a Boolean value definition
 - (a) if there is a condition of the form $q = v$ (resp., $q \neq v$) in ψ , we remove it from ψ . Then we check whether v is **true** or **false**, if not, we report failure. Otherwise, we guess atoms e_1, \dots, e_n appearing in d so that d evaluates to v (resp., to the negation of v) under assigning the atoms e_i the true value, and add to ψ the conditions e_1, \dots, e_n .
 - (c) analogous to 2.(c)
 - (d) analogous to 2.(d)
 - (e) analogous to 2.(e)
 4. if $d = [d_1, \dots, d_k]$,

- (a) if there is a condition of the form $q = v$ in ψ , we remove it from ψ , check whether v is of the form $[v_1, \dots, v_k]$, if not, we report failure. Otherwise we guess the pairs (d_i, v_j) , and for each pair (d_i, v_j) we break it down and proceed similarly the case as if we had q'/d_i and a condition $q' = v_j$ in ψ .
 - (b) if there is a condition of the form $q \neq v$ in ψ , we remove it from ψ , check whether $v \neq d$, and if not, we report failure.
 - (c) if there is a condition of the form $p' = v'$ in ψ , for a prefix p' of q , we extract the value v for p (it will be the value of the subtree in v reachable by path q' such that $p'.q' = q$) if it is possible and proceed as in (a), otherwise we report a failure.
 - (d) if there is a condition of the form $q.p' = v'$ or $q.p' \neq v'$ in ψ , we guess d_i , extract the value definition d' from d_i reachable by path p' if it is possible and proceed as in the case $q.p'/d'$, otherwise we report a failure.
 - (e) if there is a condition of the form $q.i = v'$ in ψ , we guess d_i and proceed similarly to the case as if we had q'/d_i and a condition $q' = v'$ in ψ .
5. if d is a conditional value definition ($c?d_1:d_2$), we guess atoms e_1, \dots, e_n appearing in c so that $\psi \cup \{e_1, \dots, e_n\}$ is consistent, and if c evaluates to true under assigning the atoms e_i the true value, then we consider the inductive case when d is d_1 , otherwise when d is d_2 . In any case, we add to ψ the conditions e_i .

Then we check whether the new conditions ψ' are consistent. If not, we report failure. Otherwise we check whether there is a tree satisfying ψ' in F .

- $s = \gamma_{a_1/b_1, \dots, a_n/b_n}$. If there is a condition $_id \neq \text{null}$ in ψ , we report failure. Otherwise, we remove all conditions on $_id$ from ψ , and for each a_j/b_j we proceed as follows:
 - if ψ contains a condition of the form $a_j = []$, then we check that there is no tree satisfying $\exists b_j$ in F .
 - if ψ contains a condition of the form $a_j \neq []$, then we check that there is a tree satisfying $\exists b_j$ in F .
 - if ψ contains a condition of the form $a_j = [v_1, \dots, v_k]$, $k > 0$, then for each $i = 1, \dots, k$, we check whether each tree in F satisfies $b_j \in \{v_1, \dots, v_k\}$.
 - if ψ contains a condition of the form $a_j \neq [v_1, \dots, v_k]$, $k > 0$, then we guess either “subset” or “superset”, in the former case we guess a subset $[u_1, \dots, u_m]$ of $[v_1, \dots, v_k]$ and check whether each tree in F satisfies $b_j \in \{u_1, \dots, u_m\}$ if $m > 0$, or whether each tree in F satisfies $\neg \exists b_j$ if $m = 0$; in the latter case we check whether there is a tree satisfying $\{(b_j \neq v_1), \dots, (b_j \neq v_k)\}$ in F .
 - if ψ contains a condition of the form $a_j = v$ for a non-array value, then we report failure.
 - if ψ contains a condition of the form $a_j.i = v$, then we replace it by $b_j = v$ and check whether there is a tree with the new conditions ψ' in F .
 - if ψ contains a condition $\exists a_j$, then it is satisfied and can be removed. We check whether there is a tree in F .
- $s = \gamma_{g_1/y_1, \dots, g_m/y_m; a_1/b_1, \dots, a_n/b_n}$. We replace all conditions of the form $_id.g_i = v$ in ψ by $y_i = v$. For each a_j/b_j by analogy with group by **null**, where also need to take into account conditions on $_id.g_i$.
- $s = \lambda_p^{p_1=C_2 \cdot p_2}$.
 - if there is a condition of the form $p = v$ in ψ , we remove it from ψ and
 - * if v is not an array we report a failure.
 - * if $v = []$
 - if there is a condition of the form $p_1 = v'$ in ψ , we check whether there is a tree with $p_2 = v'$ in $D.C_2$. If yes, we report failure. Otherwise, we check whether there is a tree with the new conditions ψ' in F .
 - otherwise, let v_1, \dots, v_k be all values of p_2 in $D.C_2$. We add conditions $p_1 \neq v_i$, for $i = 1, \dots, k$, to ψ , and check whether there is a tree satisfying the new conditions ψ' in F .
 - * otherwise, $v = [v_1, \dots, v_k]$, $k > 0$, and we check whether there are trees $\text{tree}(v_1), \dots, \text{tree}(v_k)$ in $D.C_2$. If not, we report failure. If yes, we check that the trees agree on the value v'' of p_2 . If yes,
 - if there is a condition of the form $p_1 = v'$ in ψ : if $v' \neq v''$, we report failure, otherwise we check whether there is a tree with the new conditions ψ' in F .
 - otherwise we add a condition $p_1 = v''$ to ψ and check whether there is a tree with the new conditions ψ' in F .
 - if ψ contains $\exists p$, we remove it and check whether there is a tree satisfying the new conditions ψ' in F .

Once we reach the first stage, then we directly check whether there is a tree in $D.C$ satisfying the conditions, or whether all trees in $D.C$ satisfy the conditions.

By analysing how we deal with various stages, we can see that both branching and alternations occur only because of the group stages. The overall algorithm works in alternating exponential time with a polynomial (actually, linear) number of alternations: the “depth” of the checks is given by the number of stages, the branching and the number of alternations are bounded by the size of q . \square

LEMMA 6.2. *Boolean query evaluation for \mathcal{M}^M queries is LOGSPACE-complete in combined complexity.*

PROOF. First, we prove the upper bound. Let D be a MongoDB database, and q an \mathcal{M}^M query of the form $C \triangleright \mu_\varphi$, where φ is a criterion. We can view φ as a Boolean formula constructed using the connectors \wedge, \vee and \neg starting from the atoms of the form $(p \text{ op } v)$ and $\exists p$, where p is a path, v a literal value, and op is a comparison operator. Given a tree t and an atom α of the above form, we can check in LOGSPACE whether $t \models \alpha$: for each node x in t , we can check in LOGSPACE if $\text{path}(x, t) = p$ and we can check in LOGSPACE if $L_n(x) = v$.

Now, we define a LOGSPACE reduction from the problem of whether $\text{ans}_{\text{mo}}(q, D) \neq \emptyset$ to the problem of determining the truth value of a variable-free Boolean formula, known to be ALOGTIME-complete [6]. We construct a Boolean formula ψ as the disjunction of φ_t for each $t \in D.C$, where φ_t is a copy of φ , where each atom α is substituted with 1 if $t \models \alpha$ and with 0, otherwise. Then $\text{ans}_{\text{mo}}(q, D) \neq \emptyset$ iff the value of ψ is true.

We show the lower bound by NC^1 reduction from the directed forest accessibility (DFA) problem known to be complete for LOGSPACE under NC^1 reducibility [11]. The DFA problem is, given an acyclic directed graph G of outdegree zero or one, nodes u and v , to decide whether there is a directed path from u to v .

Let $G = (V, T)$, $u, v \in V$ such that G has precisely two weakly connected components, u has indegree 0 and v has outdegree 0: the lower bound still holds in this case. Let v' be the other vertex in G with outdegree 0. We construct a tree $t = (N, E, L_n, L_e)$ and a path p such that $t \models (\exists p)$ iff there is a directed path from u to v in G . We add a fresh node r that will be the root of the tree with two children v and v' , and a fresh node l that will be the only child of u , also we invert all edges in G : $N = V \cup \{r, l\}$, $E = T^{-1} \cup \{(r, v), (r, v'), (u, l)\}$. Then we set $L_e(r, v) = a$, $L_e(r, v') = c$, $L_e(u, l) = b$, and the rest of the edges is labeled by index 0. The node labels are set as $L_n(r) = \langle \{\} \rangle$, $L_n(u) = \langle \{\} \rangle$ and the rest of the nodes are labeled with $\langle \{\} \rangle$.

Now, the obtained tree t is not a valid tree according to our definition of a tree, as the children of array nodes are not labeled by distinct indexes. However, by inspecting the semantics of $\llbracket p \rrbracket^t$, we see that $t \models (\exists p)$ iff $t' \models (\exists p)$, where t' is the version of t with all distinct indexes. Thus, we obtain that $t \models (\exists a.b)$ iff there is a directed path from u to v in G . \square

The project operator allows one to create new values by duplicating the existing ones; hence, it can make trees grow exponentially in the size of the query, and similarly with the group operator. Nevertheless, we can still check whether the answer to a query is non-empty in polynomial time by reusing the “old” tree nodes when it is necessary to duplicate values.

LEMMA 6.3. *Query evaluation for $\mathcal{M}^{\text{MPGL}}$ queries is PTIME-complete.*

LEMMA D.3. *The query emptiness problem for \mathcal{M}^{MP} queries is PTIME-hard in combined complexity.*

PROOF. The proof by a straightforward reduction from the Circuit Value problem, known to be PTIME-complete. For completeness, we provide the reduction. Given a monotone Boolean circuit C consisting of a finite set of assignments to Boolean variables X_1, \dots, X_n of the form $X_i = 0$, $X_i = 1$, $X_i = X_j \wedge X_k$, $j, k < i$, or $X_i = X_j \vee X_k$, $j, k < i$, where each X_i appears on the left-hand side of exactly one assignment, check whether the value X_n is 1 in C .

We construct a query q such that on each non-empty forest F , $F \triangleright q$ is non-empty iff the value X_n is 1 in C . We set $q = s_1 \triangleright \dots \triangleright s_n \triangleright \mu_{x_n=1}$, where for $i \in \{1, \dots, n\}$, $s_i = \rho_{x_1, \dots, x_{i-1}, x_i/\text{ass}_i}$, where $\text{ass}_i = v$, if $X_i = v$ for $v \in \{0, 1\}$, $\text{ass}_i = x_j \wedge x_k$, if $X_i = X_j \wedge X_k$, and $\text{ass}_i = x_j \vee x_k$, if $X_i = X_j \vee X_k$. \square

LEMMA D.4. *The query emptiness problem for $\mathcal{M}^{\text{MPGL}}$ queries is in PTIME in combined complexity.*

PROOF. We provide a PTIME algorithm for checking whether, given an $\mathcal{M}^{\text{MPGL}}$ q (over collection C), a forest F_0 for C , and forests $G_{C'}$ for each external collection C' used by q , $F_0 \triangleright q$ is non-empty.

The algorithm computes the result of each stage by representing the intermediate trees as DAGs in order to avoid exponential growth of trees that is possible due to multiple duplication of existing values. Suppose that $q = s_1 \triangleright \dots \triangleright s_m$. Then we compute F_1, \dots, F_m , where each F_i is a set of DAGs, and we can obtain from F_i the forest $F_0 \triangleright s_1 \triangleright \dots \triangleright s_i$ by “unravelling” each DAG into a proper tree.

We are going to consider connected DAGs with labeled nodes and edges and that have only one source node, that is, one node that has no incoming edges. Similarly to trees, a DAG is a tuple (N, E, L_n, L_e) , where N is a set of nodes, E is a successor relation, $L_n : N \rightarrow V \cup \{\langle \{\} \rangle, \langle \{\} \rangle\}$, $\langle \{\} \rangle$ is a node labeling function, and $L_e : E \rightarrow K \cup I$ is an edge labeling function such that (i) (N, E) forms a DAG with a single node that has no incoming edges, (ii) a node labeled by a literal must be a node without outgoing edges, (iii) all outgoing edges of a node labeled by $\langle \{\} \rangle$ must be labeled by keys, and (iv) all outgoing edges of a node labeled by $\langle \{\} \rangle$ must be labeled by distinct indexes. Clearly, a tree is a connected DAG with a single source node. We denote the source node of a DAG t by $\text{root}(t)$. For a DAG t , the path type $\text{type}(p, t)$, the interpretation of path $\llbracket p \rrbracket^t$, satisfaction of criteria $t \models \varphi$ and value definitions $t \models d$ is defined in the same way as for trees.

First, we show, given a set F of DAGs and a stage s , how to compute the set F' of DAGs resulting from evaluating s over F .

- Suppose s is a match stage μ_φ . Then $F' = \{t \mid t \in F \text{ and } t \models \varphi\}$. Clearly, $F' \subseteq F$, hence is linear in F and s .
- Suppose s is a project stage $\rho_{p_1, \dots, p_m, q_1/d_1, \dots, q_n/d_n}$. Let $t \in F$ be a DAG. We show how to transform it into a DAG t' according to s . Initially, t' contains one fresh node r with $L_n(r) = \langle \{\} \rangle$. Then, for each $i \in \{1, \dots, n\}$, we do the following changes to t' . Suppose $q_i = k_1 \dots k_l$, we first insert into t' fresh nodes x_1, \dots, x_{l-1} and edges (x_j, x_{j+1}) with $L_e(r, x_1) = k_1$, $L_e(x_{j-1}, x_j) = k_j$, and $L_n(x) = \langle \{\} \rangle$ for $x \in \{x_1, \dots, x_{l-1}\}$. Note that here, if $l = 1$, x_{l-1} refers to r . Then, by induction on the structure of d_i we proceed as follows:

- (a) d_i is a literal value v : we insert a fresh node x_l and an edge (x_{l-1}, x_l) with $L_e(x_{l-1}, x_l) = k_l$ and $L_n(x_l) = v$.
- (b) d_i is a path reference p . If $\llbracket p \rrbracket^t = \emptyset$, we remove from t' all nodes x_1, \dots, x_{l-1} (and edges) inserted previously. If $\llbracket p \rrbracket^t = 1$, let $x_p \in \llbracket p \rrbracket^t$: we add to t' the node x_p and its label, an edge (x_{l-1}, x_p) with $L_e(x_{l-1}, x_p) = k_l$, and copy all other nodes (hence the edges and labels) reachable from x_p in t . Otherwise let $\llbracket p \rrbracket^t = \{y_1, \dots, y_m\}$: we insert into t' a fresh node x_l with $L_n(x_l) = \text{'[]'}$, edges $(x_{l-1}, x_l), (x_l, y_1), \dots, (x_l, y_m)$, with $L_e(x_{l-1}, x_l) = k_l$ and $L_e(x_l, y_j) = j - 1$, and copy all other nodes (and edges and labels) reachable from y_j in t .
- (c) if d_i is a Boolean value definition, let v_b be the Boolean value of $t \models d_i$. We proceed as in the case d_i is a literal value v_b .
- (d) if d_i is a conditional value definition $(d?e_1:e_2)$, then whenever $t \models d$, we proceed as in the case d_i is e_1 , otherwise as in the case d_i is e_2 .
- (e) if d_i is an array definition $[e_1, \dots, e_m]$, then we insert into t' a fresh node x_l with $L_n(x_l) = \text{'[]'}$ and an edge (x_{l-1}, x_l) with $L_e(x_{l-1}, x_l) = k_l$. Further, for each e_j , let y_j be the node defined according to the structure of e_j and the cases above (e.g., if e_j is a literal value, then y_j is a fresh node, and if e_j is a path reference, it is an already existing in t node). We add an edge (x_l, y_j) with $L_e(x_l, y_j) = j - 1$. Note that if e_j is a path reference and this path does not exist in t , then it is equivalent to e_j being null.

Thus, we have constructed the DAG t' with a single source node. The size of t' has grown at most linearly in the size of t and s . In the resulting set F' , each DAG is obtained from exactly one DAG in F .

- Suppose s is a group stage $\gamma_{g_1/y_1, \dots, g_n/y_n; a_1/b_1, \dots, a_m/b_m}$. If $n > 1$, let F_1 be a subset of F such that
 - (\star) there exist indexes i_1, \dots, i_k , $k \leq n$, and values v_{i_1}, \dots, v_{i_k} , such that for each DAG $t \in F_1$ the following holds: $t \models (\exists y_i)$ and $t \models (y_i = v_i)$ for $i \in \{i_1, \dots, i_k\}$ and $t \models \neg(\exists y_i)$ for $i \in \{1, \dots, n\} \setminus \{i_1, \dots, i_k\}$.

We show how to transform F_1 into a DAG t_{F_1} . Initially, t_{F_1} contains two fresh nodes r and x_0 with $L_n(r) = L_n(x_0) = \text{'[]'}$, and an edge (r, x_0) with $L_e(r, x_0) = _id$. We now show how t_{F_1} is built.

First, for each $i \in \{i_1, \dots, i_k\}$, we proceed as follows. Fix a tree $t \in F_1$. Suppose $g_i = k_1 \dots k_l$, we insert into t_{F_1} fresh nodes x_1, \dots, x_{l-1} with $L_n(x) = \text{'[]'}$ for $x \in \{x_1, \dots, x_{l-1}\}$, and edges (x_j, x_{j+1}) with $L_e(x_0, x_1) = k_1$, $L_e(x_{j-1}, x_j) = k_j$. Then if $\llbracket y_i \rrbracket^t = 1$, let $x_y \in \llbracket y_i \rrbracket^t$: we add to t_{F_1} an edge (x_{l-1}, x_y) with $L_e(x_{l-1}, x_y) = k_l$, and copy all other nodes (hence the edges and labels) reachable from x_y in t . Otherwise let $\llbracket y_i \rrbracket^t = \{z_1, \dots, z_h\}$: we insert into t_{F_1} a fresh node x_l with $L_n(x_l) = \text{'[]'}$ and edges $(x_{l-1}, x_l), (x_l, z_1), \dots, (x_l, z_h)$, with $L_e(x_{l-1}, x_l) = k_l$ and $L_e(x_l, z_j) = j - 1$, and copy all other nodes (hence the edges and labels) reachable from z_j in t .

Second, for each $i = [1..m]$, we proceed as follows. We insert into t_{F_1} a fresh node x with $L_n(x) = \text{'[]'}$ and an edge (r, x) with $L_e(r, x) = a_i$. Now, for each DAG $t \in F_1$, we insert an element to the array rooted at x as follows: if $\llbracket b_i \rrbracket^t = \emptyset$, then we do not insert anything into t_{F_1} ; if $\llbracket b_i \rrbracket^t = 1$, let $z \in \llbracket b_i \rrbracket^t$, we add to t_{F_1} an edge (x, z) with $L_e(x, z)$ being the index of the new element, and copy all other nodes (hence the edges and labels) reachable from z in t ; otherwise let $\llbracket b_i \rrbracket^t = \{z_1, \dots, z_l\}$, we insert into t_{F_1} a fresh node z with $L_n(z) = \text{'[]'}$ and edges $(x, z), (z, z_1), \dots, (z, z_l)$ with $L_e(x, z)$ being the index of the new element, $L_e(z, z_j) = j - 1$, and copy all other nodes (hence the edges and labels) reachable from z_j in t .

The resulting DAG t_{F_1} has the single source node r , and its size is linear in the size of F_1 and s . Let F_1, \dots, F_l be the partition of F into subsets satisfying (\star). Such a partition can be computed in time polynomial in F and s : for each $t \in F$, we can determine its “partition” and then group the DAGs accordingly. Then F' is obtained as $\{t_{F_1}, \dots, t_{F_l}\}$, and its size is linear in the size of F and s .

- Suppose s is a lookup stage $\lambda_p^{p_1=C.p_2}$, and G is the forest for C . Let $t \in F$ with the source node x_0 , we show how to transform it into a DAG t' according to s . Initially t' coincides with t . Suppose $p = k_1 \dots k_l$, we insert into t' fresh nodes x_1, \dots, x_l with $L_n(x) = \text{'[]'}$ for $x \in \{x_1, \dots, x_{l-1}\}$, $L_n(x_l) = \text{'[]'}$, and edges (x_j, x_{j+1}) with $L_e(x_{j-1}, x_j) = k_j$. Let v be the value of p_1 in t , that is $v = \text{value}(\text{subtree}(t, p_1))$, and let G_t be the subset of G such that $\text{value}(\text{subtree}(g, p_2)) = v$ for each $g \in G_t$. Then, for each $g \in G_t$, let x_g be the root of g : we add to t' an edge (x_l, x_g) with $L_e(x_l, x_g)$ being the consecutive index, and copy the whole tree g to t' .

The resulting DAG t' is linear in the size of F , G and s .

Now, we obtain that for a query $q = s_1 \triangleright \dots \triangleright s_m$ and an input forest F_0 , each set of DAGs F_i , $i = [1..m]$ computed from F_{i-1} and s_i is linear in the size of F_{i-1} and s_i , therefore F_m is polynomial in the size of F_0 and q . It should be clear that $F_0 \triangleright q$ is non-empty iff F_m is non-empty. \square

Next, we show that adding unwind causes the loss tractability, while project and lookup do not add complexity.

LEMMA 6.5. *Boolean query evaluation for \mathcal{M}^{MU} and \mathcal{M}^{MUL} queries is NP-complete in combined complexity.*

LEMMA D.5. *Boolean query evaluation for \mathcal{M}^{MU} is NP-hard in combined complexity.*

PROOF. We prove the lower bound by reduction from the Boolean satisfiability problem. Let φ be a Boolean formula over n variables x_1, \dots, x_n . We fix a collection name C , and construct a collection F for C and an \mathcal{M}^{MU} query q such that $\text{ans}_{\text{mo}}(q, F)$ is non-empty iff φ is satisfiable.

F contains a single document d of the form $\{\{ "x1": [\text{true}, \text{false}], \dots, "xn": [\text{true}, \text{false}] \}\}$, and q is the query: $C \triangleright \omega_{x1} \triangleright \dots \triangleright \omega_{xn} \triangleright \mu_\varphi$, denoted q_{NP} , where φ can be viewed as a criterion. \square

COROLLARY D.6. *The query emptiness problem for \mathcal{M}^{MUP} queries is NP-hard in query complexity.*

PROOF. Since it is possible to use project to create copies of arrays, we can modify the above reduction so that F contains a single document of the form $\{\{ "values": [\text{true}, \text{false}] \}\}$, and $q = C \triangleright \rho_{x1/values, \dots, xn/values} \triangleright q_{NP}$. \square

COROLLARY D.7. *The query emptiness problem for \mathcal{M}^{MUL} queries is NP-hard in query complexity.*

PROOF. Now, we can use lookup to create copies of arrays. In this case again, F contains two documents of the form $\{\{ "values": \text{true} \}\}$ and $\{\{ "values": \text{false} \}\}$. The query is as follows: $q = C \triangleright \lambda_{x1}^{\text{dummy}=C.\text{dummy}} \triangleright \dots \triangleright \lambda_{xn}^{\text{dummy}=C.\text{dummy}} \triangleright \omega_{xn} \triangleright \dots \triangleright \omega_{xn} \triangleright \mu_{\varphi'}$, where φ' is the variant of φ where each variable x is replaced by $x.\text{values}$. \square

LEMMA D.8. *Boolean query evaluation for \mathcal{M}^{MUP} is in NP in combined complexity.*

PROOF. We modify the PTIME algorithm for \mathcal{M}^{MUGL} as follows. Given an \mathcal{M}^{MUP} q (over collection C), a forest F_0 for C , and forests $G_{C'}$ for each external collection C' used by q , we compute in non-deterministic polynomial time $F_0 \triangleright q$ and check whether the result is empty or not.

We only show how to compute the set F' of DAGs resulting from evaluating an unwind stage s over a set F of DAGs.

- Suppose $s = \omega_p$. Let $t \in F$, we show how to transform it into F_t , which is either the empty set or a singleton set $\{t'\}$, for a DAG t' . If p is first level array in t , let $\{x_a\} = \llbracket p \rrbracket^t$, and $\{x_1, \dots, x_n\}$ all nodes such that (x_a, x_i) are edges in t . If $n = 0$, then $F_t = \emptyset$. Otherwise, we guess $k \in \{1, \dots, n\}$ and $F_t = \{t'\}$. Initially, the new DAG t' coincides with t but on the nodes reachable from x_a . If $L_n(x_k) = \ell$ in t , then $L_n(x_a) = \ell$ in t' and we add to t' the edges (x_a, y) such that (x_k, y) is in t and copy to t' all other nodes (hence the edges and labels) reachable from y in t .
- Suppose $s = \omega_p^+$. The difference with the previous stage is that if $n = 0$, then $F_t = \{t\}$.

F' is obtained as $\bigcup_{t \in F} F_t$. Clearly, F' is linear in the size of F and s .

As a query contains a linear number of unwind stages, our algorithm requires to do a linear number of guesses (of polynomial size), and the whole computation runs in polynomial time. \square

To conclude, we also show that evaluation of \mathcal{M}^{MP} queries with additional array operators *filter*, *map* and *setUnion* is NP-hard in query complexity. The map operator $m_d(p)$ allows to transform each element inside an array p according to the new definition d , and the filter operator $f_d(p)$ filters the elements of an array p that satisfy d :

$$\begin{array}{ll} \{\$filter: \{ \text{input: PATHREF, as: PATH, cond: VALUEDEF} \}\} & d ::= f_d(p) \\ \{\$map: \{ \text{input: PATHREF, as: PATH, in: VALUEDEF} \}\} & | m_d(p) \\ \{\$setUnion: [LIST<VALUEDEF>]\} & | d_1 \cup d_2 \end{array}$$

LEMMA D.9. *The query emptiness problem for \mathcal{M}^{MP} queries with filter, map and set union operators is NP-hard in query complexity.*

PROOF. Proof by reduction from the Boolean satisfiability problem. Let φ be a Boolean formula over n variables $x1, \dots, xn$. We construct a query q such that for each non-empty forest F , $F \triangleright q$ is non-empty iff φ is satisfiable.

$$q = \rho_{a0/\{x1=0\}, a1/\{x1=1\}} \triangleright \rho_a/[a0, a1] \triangleright \quad (a_1)$$

$$\rho_{a0/m_{\{x1/ax1, x2/0\}}(a), a1/m_{\{x1/ax1, x2/1\}}(a)} \triangleright \rho_a/(a0 \cup a1) \triangleright \quad (a_2)$$

...

$$\rho_{a0/m_{\{x1/ax1, \dots, x(n-1)/ax(n-1), xn/0\}}(a), a1/m_{\{x1/ax1, \dots, x(n-1)/ax(n-1), xn/1\}}(a)} \triangleright \rho_a/(a0 \cup a1) \triangleright \quad (a_n)$$

$$\rho_{\text{assignments}/f_\varphi(a)} \triangleright \quad (\text{filter})$$

$$\mu_{\text{assignments} \neq []}$$

The stages (a_1) to (a_n) construct an array a of 2^n elements, where each element is an object encoding an assignment to the variables $x1, \dots, xn$. In the stage (a_i) , the map operator is used to extend each current element with the an assignment to the variable x_i . The (filter) stage then uses the filter operator to check for each element of the big array, whether it is a satisfying assignment, and if not, it is removed from the array. Finally, match will check that the resulting array is non-empty. If it is the case, then we have a satisfying assignment. All satisfying assignments will be stored in a . \square